[] | *neorv32_logo_riscv.png[pdfwidth=6.25in,align=center]*

The NEORV32 RISC-V Processor - Datasheet

The NEORV32 Community and Stephan Nolting

Version v1.11.0

Documentation

The online documentation of the project (a.k.a. the **data sheet**) is available on GitHub-pages: https://stnolting.github.io/neorv32/

The online documentation of the **software framework** is also available on GitHub-pages: https://stnolting.github.io/neorv32/sw/files.html

Table of Contents

Copyright by Stephan Nolting. All rights reserved.

Chapter 1. Overview

The NEORV32 RISC-V Processor is an open-source RISC-V compatible processor system that is intended as **ready-to-go** auxiliary processor within a larger SoC designs or as stand-alone custom / customizable microcontroller.

The system is highly configurable and provides optional common peripherals like embedded memories, timers, serial interfaces, general purpose IO ports and an external bus interface to connect custom IP like memories, NoCs and other peripherals. On-line and in-system debugging is supported by an OpenOCD/gdb compatible on-chip debugger accessible via JTAG.

Special focus is paid on **execution safety** to provide defined and predictable behavior at any time. Therefore, the CPU ensures that all memory access are acknowledged and no invalid/malformed instructions are executed. Whenever an unexpected situation occurs, the application code is informed via hardware exceptions.

The software framework of the processor comes with application makefiles, software libraries for all CPU and processor features, a bootloader, a runtime environment and several example programs - including a port of the CoreMark MCU benchmark and the official RISC-V architecture test suite. RISC-V GCC is used as default toolchain (prebuilt toolchains are also provided).

Check out the processor's **online User Guide** that provides hands-on tutorials to get you started.

Structure

Annotations Types

⚠	Warning
❗	Important
ℹ	Note
💡	Tip

1.1. Rationale

Why did you make this?

For me, processor and CPU architecture designs are fascinating things: they are the magic frontier where software meets hardware. This project started as something like a *journey* into this realm to understand how things actually work down on the very low level and evolved over time to a quite capable system-on-chip.

When I started to dive into the emerging RISC-V ecosystem I felt overwhelmed by the complexity. As a beginner it is hard to get an overview - especially when you want to setup a minimal platform to tinker with… Which core to use? How to get the right toolchain? What features do I need? How does booting work? How do I create an actual executable? How to get that into the hardware? How to customize things? *Where to start???*

This project aims to provide a *simple to understand* and *easy to use* yet *powerful* and *flexible* platform that targets FPGA and RISC-V beginners as well as advanced users.

Why a *soft-core* processor?

As a matter of fact soft-core processors *cannot* compete with discrete (ASIC) processors in terms of performance, energy efficiency and size. But they do fill a niche in the design space: for example, soft-core processors allow to implement the *control flow part* of certain applications (like communication protocol handling) using software like plain C. This provides high flexibility as software can be easily changed, re-compiled and re-uploaded again.

Furthermore, the concept of flexibility applies to all aspects of a soft-core processor. The user can add *exactly* the features that are required by the application: additional memories, custom interfaces, specialized co-processors and even user-defined instructions. These application-specific optimization capabilities compensate for many of the limitations of soft-core processors.

Why RISC-V?

> RISC-V is a free and open ISA enabling a new era of processor innovation through open standard collaboration.
>
> — RISC-V International, https://riscv.org/about/

Open-source is a great thing! While open-source has already become quite popular in *software*, hardware-focused projects still need to catch up. Although processors and CPUs are the heart of almost every digital system, having a true open-source platform is still a rarity. RISC-V aims to change that - and even it is *just one approach*, it helps paving the road for future development.

Furthermore, I highly appreciate the community aspect of RISC-V. The ISA and everything beyond is developed in direct contact with the community: this includes businesses and professionals but also hobbyist, amateurs and enthusiasts. Everyone can join discussions and contribute to RISC-V in their very own way.

Finally, I really like the RISC-V ISA itself. It aims to be a clean, orthogonal and "intuitive" ISA that resembles with the basic concepts of RISC: *simple yet effective*.

Yet another RISC-V core? What makes it special?

The NEORV32 is not based on another (RISC-V) core. It was build entirely from ground up just following the official ISA specs. The project does not intend to replace certain RISC-V cores or beat existing ones in terms of *performance* or *size*. It was build having a different design goal in mind.

The project aims to provide *another option* in the RISC-V / soft-core design space with a different performance vs. size trade-off and a different focus: embrace concepts like documentation, platform-independence / portability, RISC-V compatibility, extensibility & customization and - last but not least - ease of use.

Furthermore, the NEORV32 pays special focus on *execution safety* using Full Virtualization. The CPU aims to provide fall-backs for *everything that could go wrong*. This includes malformed instruction words, privilege escalations and even memory accesses that are checked for address space holes and deterministic response times of memory-mapped devices. Precise exceptions allow a defined and fully-synchronized state of the CPU at every time an in every situation.

To summarize, this project pursues the following objectives (in rough order of importance):

1. RISC-V-compliance and -compatibility
2. Functionality and features
3. Extensibility
4. Safety and security
5. Minimal area
6. Short critical paths, high operating clock
7. Simplicity / easy to understand
8. Low-power design
9. High overall performance

A multi-cycle architecture?!

The primary goal of many mainstream CPUs is pure performance. Deep pipelines and out-of-order execution are some concepts to boost performance, while also increasing complexity. In contrast, most CPUs used for teaching are single-cycle designs since they are probably the most easiest to understand. But what about something in-between?

In terms of energy, throughput, area and maximal clock frequency, multi-cycle architectures are somewhere in between single-cycle and fully-pipelined designs: they provide higher throughput and clock speed when compared to their single-cycle counterparts while having less hardware complexity (= area) and thus, less performance, then a fully-pipelined designs. So I decided to use the multi-cycle-approach because of the following reasons:

- Multi-cycle architectures are quite small! There is no need for pipeline hazard detection/resolution logic (e.g. forwarding). Furthermore, you can "re-use" parts of the core to do several tasks (e.g. the ALU is used for actual data processing and also for address generation, branch condition check and branch target computation).
- Single-cycle architectures require memories that can be read asynchronously - a thing that is not feasible to implement in real-world applications (i.e. FPGA block RAM is entirely synchronous). Furthermore, such designs usually have a very long critical path tremendously reducing maximal operating frequency.
- Pipelined designs increase performance by having several instruction "in fly" at the same time. But this also means there is some kind of "out-of-order" behavior: if an instruction at the end of the pipeline causes an exception all the instructions in earlier stages have to be invalidated. Potential architectural state changes have to be made *undone* requiring additional logic (Spectre and Meltdown…). In a multi-cycle architecture this situation cannot occur since only a single instruction is being processed ("in-fly") at a time.
- Having only a single instruction in fly does not only reduce hardware costs, it also simplifies simulation/verification/debugging, state preservation/restoring during exceptions and extensibility (no need to care about pipeline hazards) - but of course at the cost of reduced throughput.

To counteract the loss of performance implied by a *pure* multi-cycle architecture, the NEORV32 CPU uses a *mixed* approach: instruction-fetch (front-end) and instruction-execution (back-end) are de-coupled to operate independently of each other. Data is interchanged via a queue building a simple 2-stage pipeline. Each "pipeline" stage in terms is implemented as multi-cycle architecture to simplify the hardware and to provide *precise* state control (for example during exceptions).

1.2. Project Key Features

Project

- all-in-one package: **CPU** + **SoC** + **Software Framework & Tooling**
- completely described in behavioral, platform-independent VHDL - no vendor- or technology-specific primitives, attributes, macros, libraries, etc. are used at all
- all-Verilog "version" available (auto-generated by GHDL)
- extensive configuration options for adapting the processor to the requirements of the application
- highly extensible hardware - on CPU, SoC and system level
- aims to be as small as possible while being as RISC-V-compliant as possible - with a reasonable area-vs-performance trade-off

- FPGA friendly (e.g. all internal memories can be mapped to block RAM - including the register file)
- optimized for high clock frequencies to ease timing closure and integration
- from zero to *"hello world!"* - completely open source and documented
- easy to use even for FPGA/RISC-V starters – intended to *work out of the box*

NEORV32 CPU (the core)

- 32-bit RISC-V CPU
- fully compatible to the RISC-V ISA specs. - checked by the official RISCOF architecture tests
- base ISA + privileged ISA + several optional standard and custom ISA extensions
- option to add user-defined RISC-V instructions as custom ISA extension
- rich set of customization options (ISA extensions, design goal: performance / area / energy, tuning options, …)
- Full Virtualization capabilities to increase execution safety
- official RISC-V open source architecture ID

NEORV32 Processor (the SoC)

- highly-configurable full-scale microcontroller-like processor system
- based on the NEORV32 CPU
- optional standard serial interfaces (UART, TWI, SPI (host and device), 1-Wire)
- optional timers and counters (watchdog, system timer)
- optional general purpose IO and PWM; a native NeoPixel(c)-compatible smart LED interface
- optional embedded memories and caches for data, instructions and bootloader
- optional external memory interface for custom connectivity
- optional execute in-place (XIP) module to execute code directly form an external SPI flash
- optional DMA controller for CPU-independent data transfers
- optional CRC module to check data integrity
- on-chip debugger compatible with OpenOCD and GDB including hardware trigger module and optional authentication

Software framework

- GCC-based toolchain - prebuilt toolchains available; application compilation based on GNU makefiles
- internal bootloader with serial user interface (via UART)
- core libraries and HAL for high-level usage of the provided functions and peripherals
- processor-specific runtime environment and several example programs

- doxygen-based documentation of the software framework; a deployed version is available at https://stnolting.github.io/neorv32/sw/files.html
- FreeRTOS port + demos available

Extensibility and Customization

The NEORV32 processor is designed to ease customization and extensibility and provides several options for adding application-specific custom hardware modules and accelerators. The three most common options for adding custom on-chip modules are listed below.

- Processor-External Bus Interface (XBUS) to attach processor-external IP modules (memories and peripherals)
- Custom Functions Subsystem (CFS) for tightly-coupled processor-internal co-processors
- Custom Functions Unit (CFU) for custom RISC-V instructions

A more detailed comparison of the extension/customization options can be found in section Adding Custom Hardware Modules of the user guide.

1.3. Project Folder Structure

The root directory of the repository is considered the NEORV32 base or home folder (i.e. `neorv32/`).

Folder Structure

```
neorv32                  - Project home folder
 │
 ├─docs                   - Project documentation
 │  ├─datasheet            - AsciiDoc sources for the NEORV32 data sheet
 │  ├─figures              - Figures and logos
 │  ├─references           - Data sheets and RISC-V specs
 │  ├─sources              - Sources for the images in 'figures/'
 │  └─userguide            - AsciiDoc sources for the NEORV32 user guide
 │
 ├─rtl                    - VHDL sources
 │  ├─core                 - Core sources of the CPU & SoC
 │  ├─processor_templates - Pre-configured SoC wrappers
 │  ├─system_integration  - System wrappers and bridges for advanced connectivity
 │  └─test_setups          - Minimal test setup "SoCs" used in the User Guide
 │
 ├─sim                    - Simulation files
 │
 └─sw                     - Software framework
   ├─bootloader           - Sources of the processor-internal bootloader
   ├─common               - Linker script, crt0.S start-up code and central makefile
   ├─example              - Example programs for the core and the SoC modules
   │  ├─eclipse            - Pre-configured Eclipse IDE project
   │  └─...                - Several example programs
   ├─lib                  - Processor core library
   │  ├─include            - NEORV32 core library header files (*.h)
   │  └─source             - NEORV32 core library source files (*.c)
   ├─image_gen            - Helper program to generate executables & memory images
   ├─ocd_firmware         - Firmware for the on-chip debugger's "park loop"
   ├─openocd              - OpenOCD configuration files
   └─svd                  - Processor system view description file (CMSIS-SVD)
```

Copyright by Stephan Nolting. All rights reserved.

1.4. VHDL File Hierarchy

All required VHDL hardware source files are located in the project's `rtl/core` folder.

VHDL Library

All core VHDL files from the list below have to be assigned to a **new library** named `neorv32`.

Compilation Order

See section File-List Files for more information.

RTL File List (in alphabetical order)

```
rtl/core
├-neorv32_application_image.vhd - IMEM application initialization image (package)
├-neorv32_boot_rom.vhd          - Bootloader ROM
├-neorv32_bootloader_image.vhd  - Bootloader ROM memory image (package)
├-neorv32_bus.vhd               - SoC bus infrastructure modules
├-neorv32_cache.vhd             - Generic cache module
├-neorv32_clint.vhd             - Core local interruptor
├-neorv32_clockgate.vhd         - Generic clock gating switch
├-neorv32_cfs.vhd               - Custom functions subsystem
├-neorv32_cpu.vhd               - NEORV32 CPU TOP ENTITY
├-neorv32_cpu_alu.vhd           - Arithmetic/logic unit
├-neorv32_cpu_control.vhd       - CPU control, exception system and CSRs
├-neorv32_cpu_cp_bitmanip.vhd   - Bit-manipulation co-processor (B ext.)
├-neorv32_cpu_cp_cfu.vhd        - Custom instructions co-processor (Zxcfu ext.)
├-neorv32_cpu_cp_cond.vhd       - Integer conditional co-processor (Zicond ext.)
├-neorv32_cpu_cp_crypto.vhd     - Scalar cryptography co-processor (Zk*/Zbk* ext.)
├-neorv32_cpu_cp_fpu.vhd        - Floating-point co-processor (Zfinx ext.)
├-neorv32_cpu_cp_muldiv.vhd     - Mul/Div co-processor (M ext.)
├-neorv32_cpu_cp_shifter.vhd    - Bit-shift co-processor (base ISA)
├-neorv32_cpu_decompressor.vhd  - Compressed instructions decoder (C ext.)
├-neorv32_cpu_icc.vhd           - Inter-core communication unit
├-neorv32_cpu_lsu.vhd           - Load/store unit
├-neorv32_cpu_pmp.vhd           - Physical memory protection unit (Smpmp ext.)
├-neorv32_cpu_regfile.vhd       - Data register file
├-neorv32_crc.vhd               - Cyclic redundancy check unit
├-neorv32_debug_auth.vhd        - On-chip debugger: authentication module
├-neorv32_debug_dm.vhd          - On-chip debugger: debug module
├-neorv32_debug_dtm.vhd         - On-chip debugger: debug transfer module
├-neorv32_dma.vhd               - Direct memory access controller
├-neorv32_dmem.vhd              - Generic processor-internal data memory
├-neorv32_fifo.vhd              - Generic FIFO component
├-neorv32_gpio.vhd              - General purpose input/output port unit
├-neorv32_gptmr.vhd             - General purpose 32-bit timer
├-neorv32_imem.vhd              - Generic processor-internal instruction memory
```

```
  ├-neorv32_neoled.vhd            - NeoPixel (TM) compatible smart LED interface
  ├-neorv32_onewire.vhd           - One-Wire serial interface controller
  ├-neorv32_package.vhd           - Main VHDL package file
  ├-neorv32_pwm.vhd               - Pulse-width modulation controller
  ├-neorv32_sdi.vhd               - Serial data interface controller (SPI device)
  ├-neorv32_slink.vhd             - Stream link interface
  ├-neorv32_spi.vhd               - Serial peripheral interface controller (SPI host)
  ├-neorv32_sys.vhd               - System infrastructure modules
  ├-neorv32_sysinfo.vhd           - System configuration information memory
  ├-neorv32_top.vhd               - NEORV32 PROCESSOR/SOC TOP ENTITY
  ├-neorv32_trng.vhd              - True random number generator
  ├-neorv32_twd.vhd               - Two wire serial device controller
  ├-neorv32_twi.vhd               - Two wire serial interface controller
  ├-neorv32_uart.vhd              - Universal async. receiver/transmitter
  ├-neorv32_wdt.vhd               - Watchdog timer
  ├-neorv32_xbus.vhd              - External (Wishbone) bus interface gateways
  └-neorv32_xip.vhd               - Execute in place module
```

Replacing Modules for Customization or Optimization

Any module of the core can be replaced by the user for customization purpose. For example, the default IMEM and DMEM modules as well as the CPU's register file can be replaced by technology-specific primitives to optimize energy, speed and area utilization. The module, which are dedicated for customization, i.e. CFS and CFU can be replaced by user-defined modules to implement application-specific functionality.

1.4.1. File-List Files

Most of the RTL sources use **entity instantiation**. Hence, the RTL compile order might be relevant (depending on the synthesis/simulation tool. Therefore, two file-list files are provided in the `rtl` folder that list all required HDL files for the CPU core and for the entire processor and also represent their recommended compile order. These file-list files can be consumed by EDA tools to simplify project setup.

- `file_list_cpu.f` - HDL files and compile order for the CPU core; top module: `neorv32_cpu`
- `file_list_soc.f` - HDL files and compile order for the entire processor/SoC; top module: `neorv32_top`

A simple bash script `generate_file_lists.sh` is provided for regenerating the file-lists (using GHDL's *elaborate* command). This script can also be invoked using the default application makefile (see Makefile Targets).

By default, the file-list files include a **placeholder** in the path of each included hardware source file. These placeholders need to be replaced by the actual path before being used. Example:

- default: `NEORV32_RTL_PATH_PLACEHOLDER/core/neorv32_package.vhd`

- adjusted: `path/to/neorv32/rtl/core/neorv32_package.vhd`

Listing 1. Example: Processing the File-List Files in a Makefile

```
NEORV32_HOME = path/to/neorv32 ①
NEORV32_SOC_FILE = $(shell cat $(NEORV32_HOME)/rtl/file_list_soc.f) ②
NEORV32_SOC_SRCS = $(subst NEORV32_RTL_PATH_PLACEHOLDER, $(NEORV32_HOME)/rtl,
$(NEORV32_SOC_FILE)) ③
```

① Path to the NEORV32 home folder (i.e. the root folder of the GitHub repository).

② Load the content of the `file_list_soc.f` file-list into a new variable `NEORV32_SOC_FILE`.

③ Substitute the file-list file's path placeholder “NEORV32_RTL_PATH_PLACEHOLDER” by the actual path.

Listing 2. Example: Processing the File-List Files in a TCL Script

```
set file_list_file [read [open "$neorv32_home/rtl/file_list_soc.f" r]]
set file_list [string map [list "NEORV32_RTL_PATH_PLACEHOLDER" "$neorv32_home/rtl"]
$file_list_file]
puts "NEORV32 source files:"
puts $file_list
```

1.5. VHDL Coding Style

- The entire processor including the CPU core is written in platform-/technology-independent VHDL. The code makes minimal use of VHDL 2008 features to provide compatibility even for older EDA tools.
- A single package / library file (`neorv32_package.vhd`) is used to provide global defines and helper functions. The specific user-defined configuration is done entirely by the generics of the top entity.
- Internally, the generics are checked to ensure a correct configuration. Asserts and "sanity checks" are used to inform the user about the actual processor configuration and potential illegal setting.
- The code uses *entity instation* for all internal modules. However, if several "submodules" are specified within the same file *component instantiation* is used for those.
- When instantiating the processor top module (`neorv32_top.vhd`) in a custom design either entity instantiation or component instantiation can be used as the NEORV32 package file / library already provides an according component declaration.

Verilog Version

A GHDL-generated all-Verilog version of the processor is available at https://github.com/stnolting/neorv32-verilog. The provided setup generates a synthesizable Verilog netlist for a custom processor configuration.

1.6. FPGA Implementation Results

This section shows **exemplary** FPGA implementation results for the NEORV32 CPU and NEORV32 Processor modules.

The results are generated by manual synthesis runs. Hence, they might not represent the latest version of the processor.

CPU

HW version:	`1.7.8.5`
Top entity:	`rtl/core/neorv32_cpu.vhd`
FPGA:	Intel Cyclone IV E `EP4CE22F17C6`
Toolchain:	Quartus Prime Lite 21.1
Constraints:	**no timing constraints**, "balanced optimization", f_{max} from "*Slow 1200mV 0C Model*"

CPU ISA Configuration	LEs	FFs	MEM bits	DSPs	f_{max}
`rv32i_Zicsr`	1223	607	1024	0	130 MHz
`rv32i_Zicsr_Zicntr`	1578	773	1024	0	130 MHz
`rv32im_Zicsr_Zicntr`	2087	983	1024	0	130 MHz
`rv32imc_Zicsr_Zicntr`	2338	992	1024	0	130 MHz
`rv32imcb_Zicsr_Zicntr`	3175	1247	1024	0	130 MHz
`rv32imcbu_Zicsr_Zicntr`	3186	1254	1024	0	130 MHz
`rv32imcbu_Zicsr_Zicntr_Zifencei`	3187	1254	1024	0	130 MHz
`rv32imcbu_Zicsr_Zicntr_Zifencei_Zfinx`	4450	1906	1024	7	123 MHz
`rv32imcbu_Zicsr_Zicntr_Zifencei_Zfinx_DebugMode`	4825	2018	1024	7	123 MHz

Goal-Driven Optimization

The CPU provides further options to reduce the area footprint or to increase

> performance. See section Processor Top Entity - Generics for more information. Also, take a look at the User Guide section Application-Specific Processor Configuration.

1.7. CPU Performance

The performance of the NEORV32 was tested and evaluated using the Core Mark CPU benchmark. The according sources can be found in the `sw/example/coremark` folder. The resulting CoreMark score is defined as CoreMark iterations per second per MHz.

Table 1. Configuration

HW version:	`1.5.7.10`
Hardware:	32kB int. IMEM, 16kB int. DMEM, no caches, 100MHz clock
CoreMark:	2000 iterations, MEM_METHOD is MEM_STACK
Compiler:	RISCV32-GCC 10.2.0 (compiled with `march=rv32i mabi=ilp32`)
Compiler flags:	default but with `-O3`, see makefile

Table 2. CoreMark results

CPU	**CoreMark Score**	**CoreMark s/MHz**	**Average CPI**
small (`rv32i_Zicsr_Zifencei`)	33.89	**0.3389**	**4.04**
medium (`rv32imc_Zicsr_Zifencei`)	62.50	**0.6250**	**5.34**
performance (`rv32imc_Zicsr_Zifencei` + perf. options)	95.23	**0.9523**	**3.54**

The NEORV32 CPU is based on a multi-cycle architecture. Each instruction is executed in a sequence of several consecutive micro operations. The average CPI (cycles per instruction) depends on the instruction mix of a specific applications and also on the available CPU extensions. More information regarding the execution time of each implemented instruction can be found in section Instruction Sets and Extensions.

Chapter 2. NEORV32 Processor (SoC)

The NEORV32 Processor is build around the NEORV32 Central Processing Unit (CPU). Together with common peripheral interfaces and embedded memories it provides a RISC-V-based full-scale microcontroller-like SoC platform.

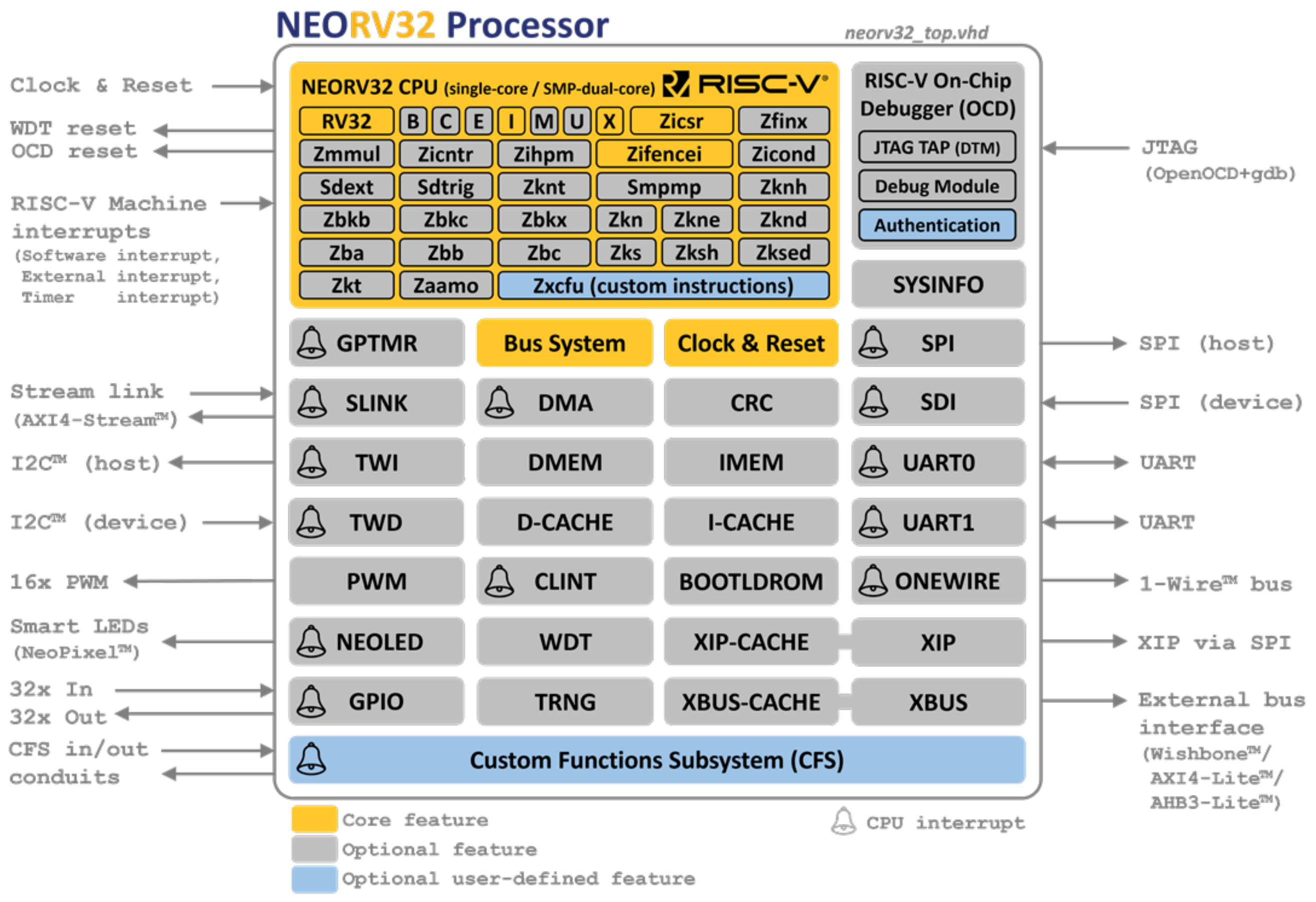


Figure 1. The NEORV32 Processor (Block Diagram)

Section Structure

- Processor Top Entity - Signals and Processor Top Entity - Generics
- Processor Clocking and Processor Reset
- Processor Interrupts
- Address Space and Boot Configuration
- Processor-Internal Modules

Key Features

- *optional* SMP Dual-Core Configuration
- *optional* processor-internal data and instruction memories (**DMEM**/**IMEM**)
- *optional* caches (**I-CACHE**, **D-CACHE**, **XIP-CACHE**, **XBUS-CACHE**)
- *optional* internal bootloader (**BOOTROM**) with UART console & SPI flash boot option

- *optional* RISC-V-compatible core local interruptor (**CLINT**)
- *optional* two independent universal asynchronous receivers and transmitters (**UART0**, **UART1**) with optional hardware flow control (RTS/CTS)
- *optional* serial peripheral interface host controller (**SPI**) with 8 dedicated CS lines
- *optional* 8-bit serial data device interface (**SDI**)
- *optional* two-wire serial interface controller (**TWI**), compatible to the I²C standard
- *optional* two-wire serial device controller (**TWD**), compatible to the I²C standard
- *optional* general purpose parallel IO port (**GPIO**), 32 inputs (interrupt capable), 32 outputs
- *optional* 32-bit external bus interface, Wishbone b4 / AXI4-Lite compatible (**XBUS**)
- *optional* watchdog timer (**WDT**)
- *optional* PWM controller with up to 16 individual channels (**PWM**)
- *optional* ring-oscillator-based true random number generator (**TRNG**)
- *optional* custom functions subsystem for custom co-processor extensions (**CFS**)
- *optional* NeoPixel™/WS2812-compatible smart LED interface (**NEOLED**)
- *optional* general purpose 32-bit timer (**GPTMR**)
- *optional* execute in-place module (**XIP**)
- *optional* 1-wire serial interface controller (**ONEWIRE**), compatible to the 1-wire standard
- *optional* autonomous direct memory access controller (**DMA**)
- *optional* stream link interface (**SLINK**), AXI4-Stream compatible
- *optional* cyclic redundancy check unit (**CRC**)
- *optional* on-chip debugger with JTAG TAP (**OCD**)
- *optional* system configuration information memory to determine hardware configuration via software (**SYSINFO**)

2.1. Processor Top Entity - Signals

The following table shows all interface signals of the processor top entity (`rtl/core/neorv32_top.vhd`). All signals are of type `std_ulogic` or `std_ulogic_vector`, respectively.

Default Values of Inputs

All *optional* input signals provide default values in case they are not explicitly assigned during instantiation. The weak driver strengths of VHDL (`'L'` and `'H'`) are used to model a pull-down or pull-up resistor.

Variable-Sized Ports

Some peripherals allow to configure the number of channels to-be-implemented by a generic (for example the number of PWM channels). The according input/output signals have a fixed sized regardless of the actually configured amount of channels. If less than the maximum number of channels is configured, only the LSB-aligned channels are used: in case of an *input port* the remaining bits/channels are left unconnected; in case of an *output port* the remaining bits/channels are hardwired to zero.

Tri-State Interfaces

Some interfaces (like the TWI, the TWD and the 1-Wire bus) require explicit tri-state drivers in the final top module.

Input/Output Registers

By default all output signals are driven by register and all input signals are synchronized into the processor's clock domain also using registers. However, for ASIC implementations it is recommended to add another register state to all inputs and output so the synthesis tool can insert an explicit IO (boundary) scan chain.

Table 3. NEORV32 Processor Signal List

Name	Width	Direction	Default	Description
Global Control (Processor Clocking and Processor Reset)				
`clk_i`	1	in	none	global clock line, all registers triggering on rising edge
`rstn_i`	1	in	none	global reset, asynchronous, **low-active**
`rstn_ocd_o`	1	out	none	Watchdog Timer (WDT) reset output, synchronous, **low-active**
`rstn_wdt_o`	1	out	none	On-Chip Debugger (OCD) reset output, synchronous, **low-active**
JTAG Access Port for On-Chip Debugger (OCD)				
`jtag_tck_i`	1	in	`'L'`	serial clock

Copyright by Stephan Nolting. All rights reserved.

Name	Width	Direction	Default	Description
`jtag_tdi_i`	1	in	`'L'`	serial data input
`jtag_tdo_o`	1	out	-	serial data output
`jtag_tms_i`	1	in	`'L'`	mode select
Processor-External Bus Interface (XBUS)				
`xbus_adr_o`	32	out	-	destination address
`xbus_dat_o`	32	out	-	read data
`xbus_tag_o`	3	out	-	access tag
`xbus_we_o`	1	out	-	write enable ('0' = read transfer)
`xbus_sel_o`	4	out	-	byte enable
`xbus_stb_o`	1	out	-	strobe
`xbus_cyc_o`	1	out	-	valid cycle
`xbus_dat_i`	32	in	`'L'`	write data
`xbus_ack_i`	1	in	`'L'`	transfer acknowledge
`xbus_err_i`	1	in	`'L'`	transfer error
Stream Link Interface (SLINK)				
`slink_rx_dat_i`	32	in	`'L'`	RX data
`slink_rx_src_i`	4	in	`'L'`	RX source routing information
`slink_rx_val_i`	1	in	`'L'`	RX data valid
`slink_rx_lst_i`	1	in	`'L'`	RX last element of stream
`slink_rx_rdy_o`	1	out	-	RX ready to receive
`slink_tx_dat_o`	32	out	-	TX data
`slink_tx_dst_o`	4	out	-	TX destination routing information
`slink_tx_val_o`	1	out	-	TX data valid
`slink_tx_lst_o`	1	out	-	TX last element of stream
`slink_tx_rdy_i`	1	in	`'L'`	TX allowed to send
Execute In Place Module (XIP)				
`xip_csn_o`	1	out	-	chip select, low-active
`xip_clk_o`	1	out	-	serial clock
`xip_dat_i`	1	in	`'L'`	serial data input
`xip_dat_o`	1	out	-	serial data output
General Purpose Input and Output Port (GPIO)				
`gpio_o`	32	out	-	general purpose parallel output

Name	Width	Direction	Default	Description
`gpio_i`	32	in	`'L'`	general purpose parallel input (interrupt-capable)
Primary Universal Asynchronous Receiver and Transmitter (UART0)				
`uart0_txd_o`	1	out	-	serial transmitter
`uart0_rxd_i`	1	in	`'L'`	serial receiver
`uart0_rts_o`	1	out	-	RX ready to receive new char
`uart0_cts_i`	1	in	`'L'`	TX allowed to start sending, low-active
Secondary Universal Asynchronous Receiver and Transmitter (UART1)				
`uart1_txd_o`	1	out	-	serial transmitter
`uart1_rxd_i`	1	in	`'L'`	serial receiver
`uart1_rts_o`	1	out	-	RX ready to receive new char
`uart1_cts_i`	1	in	`'L'`	TX allowed to start sending, low-active
Serial Peripheral Interface Controller (SPI)				
`spi_clk_o`	1	out	-	controller clock line
`spi_dat_o`	1	out	-	serial data output
`spi_dat_i`	1	in	`'L'`	serial data input
`spi_csn_o`	8	out	-	select (low-active)
Serial Data Interface Controller (SDI)				
`sdi_clk_i`	1	in	`'L'`	controller clock line
`sdi_dat_o`	1	out	-	serial data output
`sdi_dat_i`	1	in	`'L'`	serial data input
`sdi_csn_i`	1	in	`'H'`	chip select, low-active
Two-Wire Serial Interface Controller (TWI)				
`twi_sda_i`	1	in	`'H'`	serial data line sense input
`twi_sda_o`	1	out	-	serial data line output (pull low only)
`twi_scl_i`	1	in	`'H'`	serial clock line sense input
`twi_scl_o`	1	out	-	serial clock line output (pull low only)
Two-Wire Serial Device Controller (TWD)				
`twd_sda_i`	1	in	`'H'`	serial data line sense input
`twd_sda_o`	1	out	-	serial data line output (pull low only)
`twd_scl_i`	1	in	`'H'`	serial clock line sense input
`twd_scl_o`	1	out	-	serial clock line output (pull low only)

Name	Width	Direction	Default	Description
One-Wire Serial Interface Controller (ONEWIRE)				
`onewire_i`	1	in	`'H'`	1-wire bus sense input
`onewire_o`	1	out	-	1-wire bus output (pull low only)
Pulse-Width Modulation Controller (PWM)				
`pwm_o`	16	out	-	pulse-width modulated channels
Custom Functions Subsystem (CFS)				
`cfs_in_i`	32	in	`'L'`	custom CFS input signal conduit
`cfs_out_o`	32	out	-	custom CFS output signal conduit
Smart LED Interface (NEOLED)				
`neoled_o`	1	out	-	asynchronous serial data output
Core Local Interruptor (CLINT)				
`mtime_time_o`	64	out	-	CLINT.MTIMER system time output
RISC-V Machine-Mode Processor Interrupts				
`mtime_irq_i`	1	in	`'L'`	machine timer interrupt (RISC-V), high-level-active; for chip-internal usage only
`msw_irq_i`	1	in	`'L'`	machine software interrupt (RISC-V), high-level-active; for chip-internal usage only
`mext_irq_i`	1	in	`'L'`	machine external interrupt (RISC-V), high-level-active; for chip-internal usage only

2.2. Processor Top Entity - Generics

This section lists all configuration generics of the NEORV32 processor top entity (`rtl/neorv32_top.vhd`). These generics allow to configure the system according to your needs. The generics are used to control implementation of certain CPU extensions and peripheral modules and even allow to optimize the system for certain design goals like minimal area or maximum performance.

Default Values

All *optional* configuration generics provide default values in case they are not explicitly assigned during instantiation.

Software Discovery of Configuration

Software can determine the actual CPU configuration via the `misa` and `mxisa` CSRs. The Soc/Processor and can be determined via the SYSINFO memory-mapped registers.

Excluded Modules and Extensions

If optional modules (like CPU extensions or peripheral devices) are not enabled the according hardware will not be synthesized at all. Hence, the disabled modules do not increase area and power requirements and do not impact timing.

Table Abbreviations

The generic type “suv(x:y)” is an abbreviation for “std_ulogic_vector(x downto y)”.

Table 4. NEORV32 Processor Generic List

Name	Type	Default	Description
Processor Clocking			
`CLOCK_FREQUENCY`	natural	0	The clock frequency of the processor’s `clk_i` input port in Hertz (Hz).
Dual-Core Configuration			
`DUAL_CORE_EN`	boolean	false	Enable the SMP dual-core configuration.
Core Identification			
`JEDEC_ID`	suv(10:0)	"00000000000"	JEDEC ID; continuation codes plus vendor ID (passed to `mvendorid` CSR and to the Debug Transport Module (DTM)).
Boot Configuration			
`BOOT_MODE_SELECT`	natural	0	Boot mode select; see Boot Configuration.
`BOOT_ADDR_CUSTOM`	suv(31:0)	x"00000000"	Custom CPU boot address (available if `BOOT_MODE_SELECT` = 1).

Copyright by Stephan Nolting. All rights reserved.

Name	Type	Default	Description
On-Chip Debugger (OCD)			
`OCD_EN`	boolean	false	Implement the on-chip debugger and the CPU debug mode.
`OCD_AUTHENTICATION`	boolean	false	Implement Debug Authentication module.
CPU Instruction Sets and Extensions			
`RISCV_ISA_C`	boolean	false	Enable `C` ISA Extension (compressed instructions).
`RISCV_ISA_E`	boolean	false	Enable `E` ISA Extension (reduced register file size).
`RISCV_ISA_M`	boolean	false	Enable `M` ISA Extension (hardware-based integer multiplication and division).
`RISCV_ISA_U`	boolean	false	Enable `U` ISA Extension (less-privileged user mode).
`RISCV_ISA_Zaamo`	boolean	false	Enable `Zaamo` ISA Extension (atomic memory operations).
`RISCV_ISA_Zba`	boolean	false	Enable `Zba` ISA Extension (shifted-add bit-manipulation instructions).
`RISCV_ISA_Zbb`	boolean	false	Enable `Zbb` ISA Extension (basic bit-manipulation instructions).
`RISCV_ISA_Zbkb`	boolean	false	Enable `Zbkb` ISA Extension (scalar cryptography bit manipulation instructions).
`RISCV_ISA_Zbkc`	boolean	false	Enable `Zbkc` ISA Extension (scalar cryptography carry-less multiplication instructions).
`RISCV_ISA_Zbkx`	boolean	false	Enable `Zbkx` ISA Extension (scalar cryptography crossbar permutations).
`RISCV_ISA_Zbs`	boolean	false	Enable `Zbs` ISA Extension (single-bit bit-manipulation instructions).
`RISCV_ISA_Zfinx`	boolean	false	Enable `Zfinx` ISA Extension (single-precision floating-point unit).
`RISCV_ISA_Zicntr`	boolean	true	Enable `Zicntr` ISA Extension (CPU base counters).
`RISCV_ISA_Zicond`	boolean	false	Enable `Zicond` ISA Extension (integer conditional instructions).
`RISCV_ISA_Zihpm`	boolean	false	Enable `Zihpm` ISA Extension (hardware performance monitors).
`RISCV_ISA_Zknd`	boolean	false	Enable `Zknd` ISA Extension (scalar cryptography NIST AES decryption instructions).
`RISCV_ISA_Zkne`	boolean	false	Enable `Zkne` ISA Extension (scalar cryptography NIST AES encryption instructions).

Name	Type	Default	Description
RISCV_ISA_Zknh	boolean	false	Enable Zknh ISA Extension (scalar cryptography NIST hash instructions).
RISCV_ISA_Zksed	boolean	false	Enable Zksed ISA Extension (scalar cryptography ShangMi block cyphers).
RISCV_ISA_Zksh	boolean	false	Enable Zksh ISA Extension (scalar cryptography ShangMi hash functions).
RISCV_ISA_Zmmul	boolean	false	Enable Zmmul - ISA Extension (hardware-based integer multiplication).
RISCV_ISA_Zxcfu	boolean	false	Enable NEORV32-specific Zxcfu ISA Extension (custom RISC-V instructions).
CPU Tuning Options			
CPU_CLOCK_GATING_EN	boolean	false	Implement sleep-mode clock gating; see sections Sleep Mode and CPU Clock Gating.
CPU_FAST_MUL_EN	boolean	false	Implement fast but large full-parallel multipliers (trying to infer DSP blocks); see section CPU Arithmetic Logic Unit.
CPU_FAST_SHIFT_EN	boolean	false	Implement fast but large full-parallel barrel shifters; see section CPU Arithmetic Logic Unit.
CPU_RF_HW_RST_EN	boolean	false	Implement full hardware reset for register file (use individual FFs instead of BRAM); see section CPU Register File.
Physical Memory Protection (Smpmp ISA Extension)			
PMP_NUM_REGIONS	natural	0	Number of implemented PMP regions (0..16).
PMP_MIN_GRANULARITY	natural	4	Minimal region granularity in bytes. Has to be a power of two, min 4.
PMP_TOR_MODE_EN	boolean	true	Implement support for top-of-region (TOR) mode.
PMP_NAP_MODE_EN	boolean	true	Implement support for naturally-aligned power-of-two (NAPOT & NA4) modes.
Hardware Performance Monitors (Zihpm ISA Extension)			
HPM_NUM_CNTS	natural	0	Number of implemented hardware performance monitor counters (0..13).
HPM_CNT_WIDTH	natural	40	Total LSB-aligned size of each HPM counter. Min 0, max 64.
Internal Instruction Memory (IMEM)			
MEM_INT_IMEM_EN	boolean	false	Implement the processor-internal instruction memory.

Name	Type	Default	Description
MEM_INT_IMEM_SIZE	natural	16*1024	Size in bytes of the processor internal instruction memory (use a power of 2).
Internal Data Memory (DMEM)			
MEM_INT_DMEM_EN	boolean	false	Implement the processor-internal data memory.
MEM_INT_DMEM_SIZE	natural	8*1024	Size in bytes of the processor-internal data memory (use a power of 2).
Processor-Internal Instruction Cache (iCACHE)			
ICACHE_EN	boolean	false	Implement the instruction cache.
ICACHE_NUM_BLOCKS	natural	4	Number of blocks ("lines") Has to be a power of two.
ICACHE_BLOCK_SIZE	natural	64	Size in bytes of each block. Has to be a power of two.
Processor-Internal Data Cache (dCACHE)			
DCACHE_EN	boolean	false	Implement the data cache.
DCACHE_NUM_BLOCKS	natural	4	Number of blocks ("lines"). Has to be a power of two.
DCACHE_BLOCK_SIZE	natural	64	Size in bytes of each block. Has to be a power of two.
Processor-External Bus Interface (XBUS) (Wishbone b4 protocol)			
XBUS_EN	boolean	false	Implement the external bus interface.
XBUS_TIMEOUT	natural	255	Clock cycles after which a pending external bus access will auto-terminate and raise a bus fault exception.
XBUS_REGSTAGE_EN	boolean	false	Implement XBUS register stages to ease timing closure.
XBUS_CACHE_EN	boolean	false	Implement the external bus cache.
XBUS_CACHE_NUM_BLOCKS	natural	64	Number of blocks ("lines"). Has to be a power of two.
XBUS_CACHE_BLOCK_SIZE	natural	32	Size in bytes of each block. Has to be a power of two.
Execute In Place Module (XIP)			
XIP_EN	boolean	false	Implement the execute in-place module.
XIP_CACHE_EN	boolean	false	Implement XIP cache.
XIP_CACHE_NUM_BLOCKS	natural	8	Number of blocks in XIP cache. Has to be a power of two.

Name	Type	Default	Description
`XIP_CACHE_BLOCK_SIZE`	natural	256	Number of bytes per XIP cache block. Has to be a power of two, min 4.
			Peripheral/IO Modules
`IO_DISABLE_SYSINFO`	boolean	false	Disable System Configuration Information Memory (SYSINFO) module; □□ not recommended - for advanced users only!
`IO_GPIO_NUM`	natural	0	Number of general purpose input/output pairs of the General Purpose Input and Output Port (GPIO), max 32.
`IO_CLINT_EN`	boolean	false	Implement the Core Local Interruptor (CLINT).
`IO_UART0_EN`	boolean	false	Implement the Primary Universal Asynchronous Receiver and Transmitter (UART0).
`IO_UART0_RX_FIFO`	natural	1	UART0 RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
`IO_UART0_TX_FIFO`	natural	1	UART0 TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
`IO_UART1_EN`	boolean	false	Implement the Secondary Universal Asynchronous Receiver and Transmitter (UART1).
`IO_UART1_RX_FIFO`	natural	1	UART1 RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
`IO_UART1_TX_FIFO`	natural	1	UART1 TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
`IO_SPI_EN`	boolean	false	Implement the Serial Peripheral Interface Controller (SPI).
`IO_SPI_FIFO`	natural	1	Depth of the Serial Peripheral Interface Controller (SPI) FIFO. Has to be a power of two, min 1, max 32768.
`IO_SDI_EN`	boolean	false	Implement the Serial Data Interface Controller (SDI).
`IO_SDI_FIFO`	natural	1	Depth of the Serial Data Interface Controller (SDI) FIFO. Has to be a power of two, min 1, max 32768.
`IO_TWI_EN`	boolean	false	Implement the Two-Wire Serial Interface Controller (TWI).
`IO_TWI_FIFO`	natural	1	Depth of the Two-Wire Serial Interface Controller (TWI) FIFO. Has to be a power of two, min 1, max 32768.
`IO_TWD_EN`	boolean	false	Implement the Two-Wire Serial Device Controller (TWD).

Name	Type	Default	Description
`IO_TWD_FIFO`	natural	1	Depth of the Two-Wire Serial Device Controller (TWD) FIFO. Has to be a power of two, min 1, max 32768.
`IO_PWM_NUM_CH`	natural	0	Number of channels of the Pulse-Width Modulation Controller (PWM) to implement (0..16).
`IO_WDT_EN`	boolean	false	Implement the Watchdog Timer (WDT).
`IO_TRNG_EN`	boolean	false	Implement the True Random-Number Generator (TRNG).
`IO_TRNG_FIFO`	natural	1	Depth of the TRNG data FIFO. Has to be a power of two, min 1, max 32768.
`IO_CFS_EN`	boolean	false	Implement the Custom Functions Subsystem (CFS).
`IO_CFS_CONFIG`	suv(31:0)	x"00000000"	"Conduit" generic to pass user-defined flags to the Custom Functions Subsystem (CFS).
`IO_CFS_IN_SIZE`	natural	32	Size of the Custom Functions Subsystem (CFS) input signal conduit (`cfs_in_i`).
`IO_CFS_OUT_SIZE`	natural	32	Size of the Custom Functions Subsystem (CFS) output signal conduit (`cfs_out_o`).
`IO_NEOLED_EN`	boolean	false	Implement the Smart LED Interface (NEOLED).
`IO_NEOLED_TX_FIFO`	natural	1	TX FIFO depth of the the Smart LED Interface (NEOLED). Has to be a power of two, min 1, max 32768.
`IO_GPTMR_EN`	boolean	false	Implement the General Purpose Timer (GPTMR).
`IO_ONEWIRE_EN`	boolean	false	Implement the One-Wire Serial Interface Controller (ONEWIRE).
`IO_ONEWIRE_FIFO`	natural	1	Depth of the One-Wire Serial Interface Controller (ONEWIRE) FIFO. Has to be a power of two, min 1, max 32768.
`IO_DMA_EN`	boolean	false	Implement the Direct Memory Access Controller (DMA).
`IO_SLINK_EN`	boolean	false	Implement the Stream Link Interface (SLINK).
`IO_SLINK_RX_FIFO`	natural	1	SLINK RX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
`IO_SLINK_TX_FIFO`	natural	1	SLINK TX FIFO depth, has to be a power of two, minimum value is 1, max 32768.
`IO_CRC_EN`	boolean	false	Implement the Cyclic Redundancy Check (CRC) unit.

2.3. Processor Clocking

The processor is implemented as fully-synchronous logic design using a single clock domain that is driven entirely by the top's `clk_i` signal. This clock signal is used by all internal registers and memories. All of them trigger on the **rising edge** of this clock signal. External "clocks" like the OCD's JTAG clock or the SDI's serial clock are synchronized into the processor's clock domain before being used as "general logic signal" (and not as a dedicated clock).

CPU Clock Gating

The CPU core provides an optional clock-gating feature to switch off large parts of the core when sleep mode is entered. See section CPU Clock Gating for more information.

2.3.1. Peripheral Clocks

Many processor modules like the UARTs or the timers provide a programmable time base for operations. In order to simplify the hardware, the processor implements a global "clock generator" (`neorv32_sys.vhd`) that provides single-cycle *clock enables* for certain frequencies which are derived from the main clock. These clock enable signals are synchronous to the system's main clock. The processor modules can use these enables for sub-main-clock operations while still providing a single clock domain only.

In total, 8 sub-main-clock signals are available. All processor modules, which feature a time-based configuration, provide a programmable three-bit prescaler select in their control register to select one of the 8 available clocks. The mapping of the prescaler select bits to the according clock source is shown in the table below. Here, *f* represents the processor main clock from the top entity's `clk_i` signal.

Prescaler bits:	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting clock:	*f/2*	*f/4*	*f/8*	*f/64*	*f/128*	*f/1024*	*f/2048*	*f/4096*

Power Saving

If no peripheral modules requires a clock signal from the internal clock generator (all according modules are disabled by clearing the enable bit in the according module's control register) the generator is automatically deactivated to reduce dynamic power consumption.

Copyright by Stephan Nolting. All rights reserved.

2.4. Processor Reset

The NEORV32 processor includes a central reset sequencer (`neorv32_sys.vhd`) that handles all reset requests and controls the internal reset nets. The processor-wide reset (aka "system reset") can be triggered by any of the following sources:

- the asynchronous low-active `rstn_i` top entity input signal (External source)
- the On-Chip Debugger (OCD) (internal source)
- the Watchdog Timer (WDT) (internal source)

Processor Reset Signal

Make sure to connect the processor's reset signal `rstn_i` to a valid reset source (a button, the "locked" signal of a PLL, a dedicated reset controller, etc.).

Reset Cause

The actual reset cause can be determined via the Watchdog Timer (WDT).

If any of these sources triggers a reset, the internal system-wide reset will be active for at least 4 clock cycles ensuring a valid reset of the entire processor. This system reset is asserted *asynchronoulsy* if triggered by the external `rstn_i` signal and is asserted *synchronously* if triggered by an internal reset source. However, the system reset is always de-asserted *synchronously* at the next rising clock edge.

Internally, **all registers** that are not meant for mapping to blockRAM (like the register file) do provide a dedicated and **low-active asynchronous** hardware reset. This asynchronous reset ensures that the entire processor logic is reset to a defined state even if the main clock is not operational yet.

2.5. Processor Interrupts

The NEORV32 Processor provides several interrupt request signals (IRQs) for custom platform use.

Trigger Type

All interrupt request lines are **level-triggered and high-active**. Once set, the signal should remain high until the interrupt request is explicitly acknowledged (e.g. writing to a memory-mapped register).

2.5.1. RISC-V Standard Interrupts

The processor setup features the standard machine-level RISC-V interrupt lines for "machine timer interrupt", "machine software interrupt" and "machine external interrupt". Their usage is defined by the RISC-V privileged architecture specifications. However, bare-metal system can also repurpose these interrupts. See CPU section Traps, Exceptions and Interrupts for more information.

Top signal	Description
`mtime_irq_i`	Machine timer interrupt from *processor-external* CLINT (`MTI`). This IRQ is only available if the processor-internal Core Local Interruptor (CLINT) unit is not implemented.
`msw_irq_i`	Machine software interrupt from *processor-external* CLINT (`MSI`). This IRQ is only available if the processor-internal Core Local Interruptor (CLINT) unit is not implemented.
`mext_irq_i`	Machine external interrupt (`MEI`). This interrupt is used for any processor-external interrupt source (like a platform interrupt controller).

2.5.2. NEORV32-Specific Fast Interrupt Requests

As part of the NEORV32-specific CPU extensions, the processor core features 16 fast interrupt request signals (`FIRQ0` to `FIRQ15`) providing dedicated bits in the **`mip`** and **`mie`** CSRs and custom **`mcause`** trap codes. The FIRQ signals are reserved for *processor-internal* modules only (for example for the communication interfaces to signal "available incoming data" or "ready to send new data").

The mapping of the 16 FIRQ channels to the according processor-internal modules is shown in the following table (the channel number also corresponds to the according FIRQ priority: 0 = highest, 15 = lowest):

Table 5. NEORV32 Fast Interrupt Request (FIRQ) Mapping

Channel	Source	Description
0	TWD	TWD FIFO level interrupt
1	CFS	Custom functions subsystem (CFS) interrupt (user-defined)
2	UART0	UART0 RX FIFO level interrupt

Copyright by Stephan Nolting. All rights reserved.

Channel	Source	Description
3	UART0	UART0 TX FIFO level interrupt
4	UART1	UART1 RX FIFO level interrupt
5	UART1	UART1 TX FIFO level interrupt
6	SPI	SPI FIFO level interrupt
7	TWI	TWI FIFO level interrupt
8	GPIO	GPIO input pin(s) interrupt
9	NEOLED	NEOLED TX FIFO level interrupt
10	DMA	DMA transfer done interrupt
11	SDI	SDI FIFO level interrupt
12	GPTMR	General purpose timer interrupt
13	ONEWIRE	1-wire idle interrupt
14	SLINK	SLINK RX FIFO level interrupt
15	SLINK	SLINK TX FIFO level interrupt

2.6. Address Space

As a 32-bit architecture the NEORV32 can access a 4GB physical address space. By default, this address space is split into four main regions. All accesses to "unmapped" addresses (a.k.a. "the void") are redirected to the Processor-External Bus Interface (XBUS). For example, if the internal IMEM is disabled, the accesses to the *entire* address space between `0x00000000` and `0x7FFFFFFF` are converted into XBUS requests. If the XBUS interface is not enabled any access to the void will raise a bus error exception.

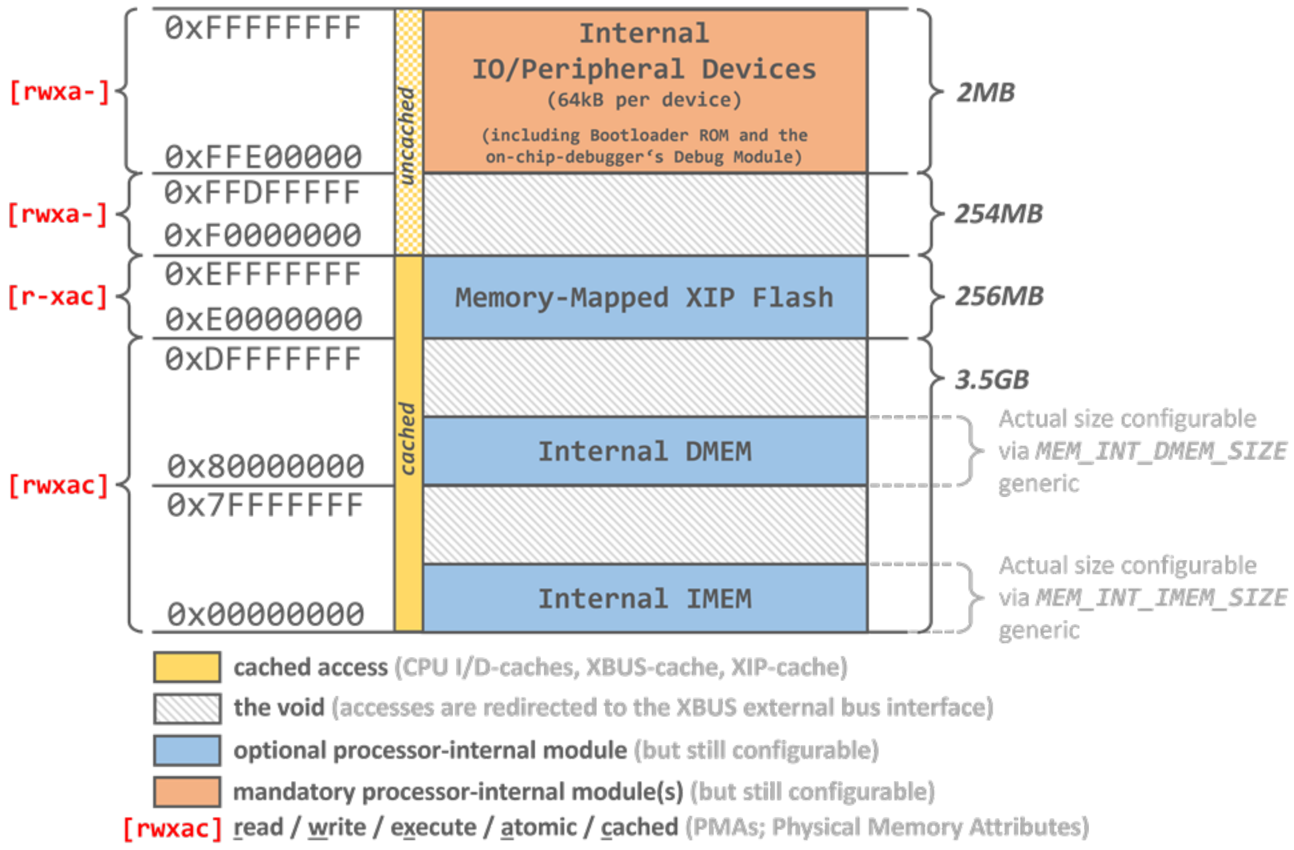

[rwxa-]
0xFFFFFFFF
0xFFE00000
Internal IO/Peripheral Devices
(64kB per device)
(including Bootloader ROM and the on-chip-debugger's Debug Module)
uncached
2MB
[rwxa-]
0xFFDFFFFF
0xF0000000
254MB
[r-xac]
0xEFFFFFFF
0xE0000000
Memory-Mapped XIP Flash
256MB
0xDFFFFFFF
0x80000000
0x7FFFFFFF
0x00000000
[rwxac]
cached
Internal DMEM
Internal IMEM
3.5GB
Actual size configurable via MEM_INT_DMEM_SIZE generic
Actual size configurable via MEM_INT_IMEM_SIZE generic
cached access (CPU I/D-caches, XBUS-cache, XIP-cache)
the void (accesses are redirected to the XBUS external bus interface)
optional processor-internal module (but still configurable)
mandatory processor-internal module(s) (but still configurable)
[rwxac] read / write / execute / atomic / cached (PMAs; Physical Memory Attributes)

Figure 2. NEORV32 Processor Address Space (Default Configuration)

Each region provides specific *physical memory attributes* ("PMAs") that define the access capabilities (`rwxac`; `r` = read access, `w` = write access, `x` - execute access, `a` = atomic access, `c` = cached CPU access).

> *Custom PMAs*
>
> Custom physical memory attributes enforced by the CPU's *physcial memory protection* (`Smpmp` ISA Extension) can be used to further constrain the physical memory attributes.

Table 6. Main Address Regions

#	Region	PMAs	Description
1	Internal IMEM address space	`rwxac`	For instructions / code and constants; mapped to the internal Instruction Memory (IMEM) if implemented.
2	Internal DMEM address space	`rwxac`	For application runtime data (heap, stack, etc.); mapped to the internal Data Memory (DMEM)) if implemented.
3	Memory-mapped XIP flash	`r-xac`	Transparent memory-mapped access to an external Execute In Place Module (XIP) SPI flash.
4	IO/peripheral address space	`rwxa-`	Processor-internal peripherals / IO devices including the Bootloader ROM (BOOTROM).
-	The "**void**"	`rwxa[c]`	Unmapped address space. All accesses to this region(s) are redirected to the Processor-External Bus Interface (XBUS) if implemented.

2.6.1. Bus System

The CPU provides individual interfaces for instruction fetch and data access. It can can access all of the 32-bit address space from each of the interface. Both of them can be equipped with optional caches (Processor-Internal Data Cache (dCACHE) and Processor-Internal Instruction Cache (iCACHE)).

The two CPU interfaces are multiplexed by a simple bus switch into a *single processor-internal bus*. Optionally, this bus is further multiplexed by another instance of the bus switch so the Direct Memory Access Controller (DMA) controller can also access the entire address space. Accesses via the resulting SoC bus are split by the Bus Gateway that redirects accesses to the according main address regions (see table above). Accesses to the processor-internal IO/peripheral devices are further redirected via a dedicated IO Switch.

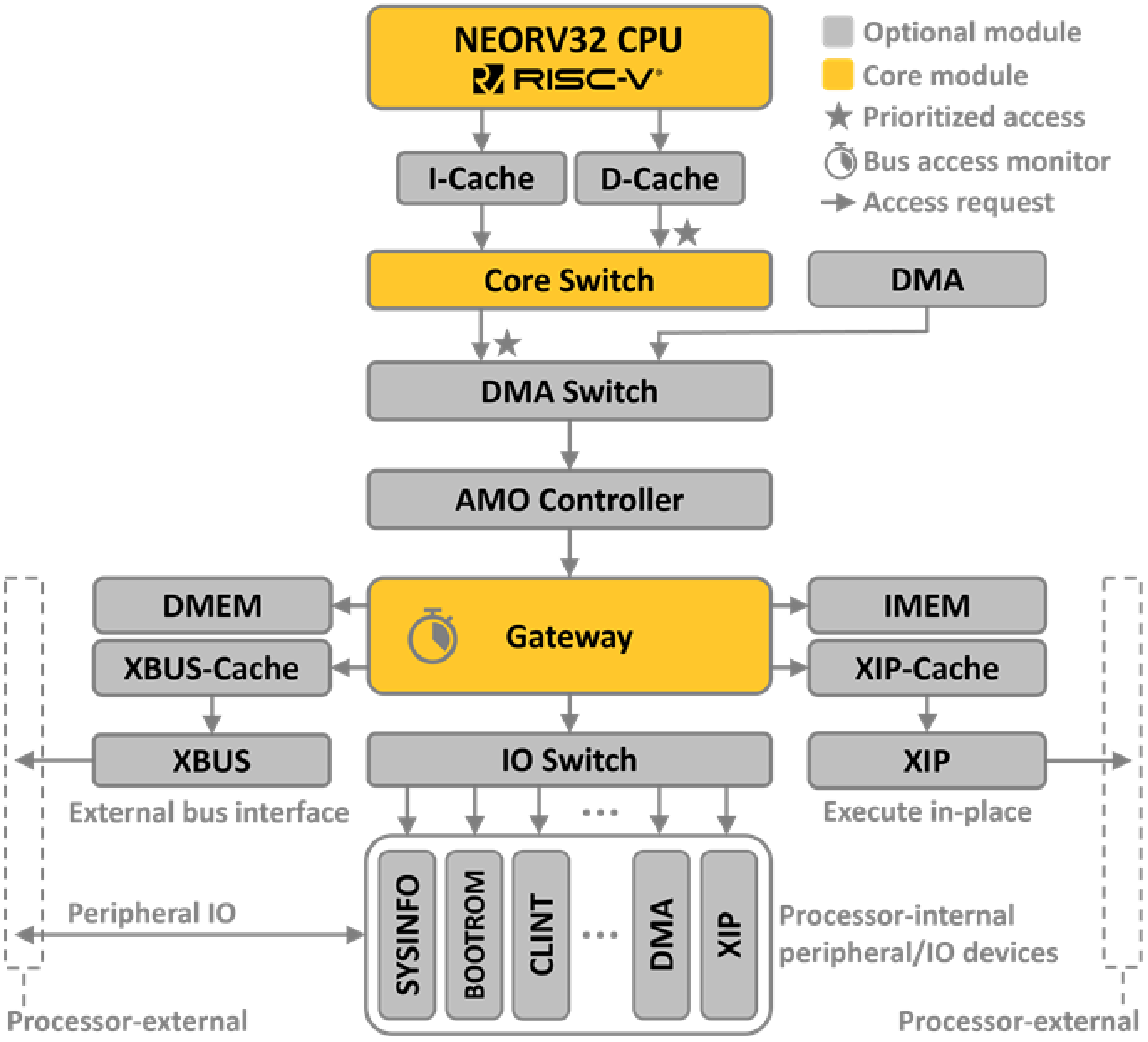


Figure 3. Processor-Internal Bus Architecture

Bus System Infrastructure

The components of the processor’s bus system infrastructure are located in `rtl/core/neorv32_bus.vhd`.

Bus Interface

See sections CPU Architecture and Bus Interface for more information regarding the CPU bus accesses.

SMP Dual-Core Configuration

The dual-core configuration adds a second CPU core complex in parallel to the first one. See section Dual-Core Configuration for more information.

2.6.2. Bus Gateway

The central bus gateway serves two purposes: it **redirects** accesses to the according modules (e.g. memory accesses vs. memory-mapped IO accesses) and also **monitors** all bus transactions. The redirection of access request is based on a customizable memory map implemented via VHDL constants in the main package file (`rtl/core/neorv323_package.vhd`):

Listing 3. Main Address Regions Configuration in the VHDL Package File

```
-- Main Address Regions ---
constant mem_imem_base_c : std_ulogic_vector(31 downto 0) := x"00000000"; -- IMEM size
via generic
constant mem_dmem_base_c : std_ulogic_vector(31 downto 0) := x"80000000"; -- DMEM size
via generic
constant mem_xip_base_c  : std_ulogic_vector(31 downto 0) := x"e0000000"; -- page (4
MSBs) only!
constant mem_xip_size_c  : natural := 256*1024*1024;
constant mem_io_base_c   : std_ulogic_vector(31 downto 0) := x"ffe00000";
constant mem_io_size_c   : natural := 32*64*1024; -- = 32 * iodev_size_c
```

Besides the redirecting of bus requests the gateway also implements a bus monitor (aka "the bus keeper") that tracks all active bus transactions to ensure *safe* and *deterministic* operations. Whenever a memory-mapped device is accessed (a real memory, a memory-mapped IO or some processor-external module) the bus monitor starts an internal countdown. The accessed module has to respond ("ACK") to the bus request within a bound **time window**. This time window is defined by a global constant in the processor's VHDL package file (`rtl/core/neorv323_package.vhd`).

Listing 4. Internal Bus Timeout Configuration

```
constant bus_timeout_c : natural := 15;
```

This constant defines the *maximum* number of cycles after which a non-responding bus request (i.e. no `ack` and no `err` signal) will time out raising a bus access fault exception. For example this can happen when accessing "address space holes" - addresses that are not mapped to any physical module. The resulting exception type corresponds to the according access type, i.e. instruction fetch access exception, load access exception or store access exception.

XIP Timeout

Accesses to the memory-mapped XIP flash (via the Execute In Place Module (XIP)) will *never* time out.

External Bus Interface Timeout

Accesses that are delegated to the external bus interface have a different maximum timeout value that is defined by an explicit specific processor generic. See section Processor-External Bus Interface (XBUS) for more information.

2.6.3. IO Switch

The IO switch further decodes the address when accessing the processor-internal IO/peripheral devices and forwards the access request to the according module. Note that a total address space size of 256 bytes is assigned to each IO module in order to simplify address decoding. The IO-specific address map is also defined in the main VHDL package file (`rtl/core/neorv323_package.vhd`).

Listing 5. Exemplary Cut-Out from the IO Address Map

```
-- IO Address Map --
constant iodev_size_c    : natural := 256; -- size of a single IO device (bytes)
constant base_io_cfs_c   : std_ulogic_vector(31 downto 0) := x"ffffeb00";
constant base_io_slink_c : std_ulogic_vector(31 downto 0) := x"ffffec00";
constant base_io_dma_c   : std_ulogic_vector(31 downto 0) := x"ffffed00";
```

2.6.4. Atomic Memory Operations Controller

The atomic memory operations (AMO) controller is responsible for handling the read-modify-write operations issued by the CPU's `Zaamo` ISA Extension. For each AMO request, the controller executes an atomic set of three operations:

Table 7. Simplified AMO Controller Operation

Step	Pseudo Code	Description
1	`tmp1 ⇐ MEM[address];`	Perform a read operation accessing the addressed memory cell and store the loaded data into an internal buffer (`tmp1`).
2	`tmp2 ⇐ tmp1 OP cpu_wdata`	The buffered data from the first step is processed using the write data provide by the CPU. The result is stored to another internal buffer (`tmp2`).
3	`MEM[address] ⇐ tmp2;` `cpu_rdata ⇐ tmp1;`	The data from the second buffer (`tmp2`) is written to the addressed memory cell. In parallel, the data from the first buffer (`tmp1` = original content of the addresses memory cell) is sent back to the requesting CPU.

Direct Access

Atomic operations **always bypass** the CPU's data cache using direct/uncached accesses. Care must be taken to maintain data Cache Coherency.

Physical Memory Attributes

Atomic memory operations can be executed for *any* address. This also includes cached memory, memory-mapped IO devices and processor-external address spaces.

The controller performs two bus transactions: a read operations and a write operation. Only the acknowledge/error handshake of the last transaction is sent back to the CPU.

As the AMO controller is the memory-nearest instance (see Bus System) the previously described set of operations cannot be interrupted. Hence, they execute in an atomic way.

2.6.5. Cache Coherency

In total the NEORV32 Processor provides up to four optional caches organized in two levels. Level-1 caches are closer to the CPU while level-2 caches are closer to main memory (however, this highly depends on the the actual cache configurations).

- The Processor-Internal Data Cache (dCACHE) (level-1)
- The Processor-Internal Instruction Cache (iCACHE) (level-1)
- The cache of the Processor-External Bus Interface (XBUS) (level-2)
- The cache of the Execute In Place Module (XIP) (level-2)

As all caches operate transparently for the software, special attention must therefore be paid to coherence. Note that coherence and cache *synchronization* is **not** performed by the hardware itself (there is no snooping implemented).

The NEORV32 uses two instructions for manual cache synchronization (both instructions are always available regardless of the actual CPU/ISA configuration):

- `fence` (`I` ISA Extension / `E` ISA Extension)
- `fence.i` (`Zifencei` ISA Extension)

By executing the "data" `fence` instruction the CPU's data cache is synchronized in four steps:

1. The CPU data cache is flushed: all local modifications are copied to the next higher memory level; this can be the XBUS cache or main memory.
2. The CPU data cache is cleared invalidating all local entries.
3. The synchronization request is sent to the next-higher memory level (for example to the XBUS cache so it can perform the same synchronization steps).
4. The CPU data cache is reloaded with up-to-date data from the next higher memory level.

By executing the "instruction" `fence.i` instruction the CPU's instruction cache is synchronized in three steps:

1. The synchronization request is sent to the next-higher memory level (for example to the XBUS cache so it can perform the same synchronization steps).
2. The CPU instruction cache is cleared invalidating all local entries.
3. The CPU instruction cache is reloaded with up-to-date data from the next higher memory level.

2.7. Boot Configuration

The NEORV32 processor provides some pre-defined boot configurations to adjust system start-up to the requirements of the application. The actual boot configuration is defined by the `BOOT_MODE_SELECT` generic (see Processor Top Entity - Generics).

Table 8. NEORV32 Boot Configurations

`BOOT_MODE_SELECT`	Name	Boot address	Description
0 (default)	Bootloader	Base of internal BOOTROM	Implement the processor-internal Bootloader ROM (BOOTROM) as pre-initialized ROM and boot from there.
1	Custom Address	`BOOT_ADDR_CUSTOM` generic	Start booting at user-defined address (`BOOT_ADDR_CUSTOM` top generic).
2	IMEM Image	Base of internal IMEM	Implement the processor-internal Instruction Memory (IMEM) as pre-initialized ROM and boot from there.

Dual-Core Boot

For the SMPA dual-core CPU configuration boot procedure see section Dual-Core Boot.

2.7.1. Booting via Bootloader

This is the most common and thus, the default boot configuration. When selected, the processor-internal Bootloader ROM (BOOTROM) is enabled. This ROM contains the executable image (`rtl/core/neorv32_bootloader_image.vhd`) of the default NEORV32 Bootloader that will be executed right after reset. The bootloader provides an interactive user console for executable upload as well as an automatic boot-configuration targeting external (SPI) memories.

If the processor-internal Instruction Memory (IMEM) is enabled it will be implemented as *blank* RAM.

2.7.2. Boot from Custom Address

This is the most flexible boot configuration as it allows the user to specify a custom boot address via the `BOOT_ADDR_CUSTOM` generic. Note that this address has to be aligned to 4-byte boundary. The processor will start executing from the defined address right after reset. For example, this boot configuration ca be used to execute a *custom bootloader* from a memory that is attached via the Processor-External Bus Interface (XBUS).

The Bootloader ROM (BOOTROM) is not enabled / implement at all. If the processor-internal Instruction Memory (IMEM) is enabled it will be implemented as *blank* RAM.

2.7.3. Boot IMEM Image

This configuration will implement the Instruction Memory (IMEM) as *pre-initialized read-only memory* (ROM). The ROM is initialized during synthesis with the according application image file (`rtl/core/neorv32_application_image.vhd`). After reset, the CPU will directly start executing this image. Since the IMEM is implemented as ROM, the executable cannot be altered at runtime at all.

The Bootloader ROM (BOOTROM) is not enabled / implement at all.

Internal IMEM is Required

This boot configuration requires the IMEM to be enabled (`MEM_INT_IMEM_EN` = true).

Simulation Setup

This boot configuration is handy for simulations as the application software is executed right away without the need for an explicit initialization / executable upload.

2.8. Processor-Internal Modules

Full-Word Write Accesses Only

All peripheral/IO devices should only be accessed in full-word mode (i.e. 32-bit). Byte or half-word (8/16-bit) write accesses might cause undefined behavior.

IO Module Address Space

Each peripheral/IO module occupies an address space of 64kB bytes. Most devices do not fully utilize this address space and will *mirror* the available memory-mapped registers across the entire 64kB address space. However, accessing memory-mapped registers other than the specified ones should be avoided.

Unimplemented Modules / Address Holes

When accessing an IO device that hast not been implemented (disabled via the according generic) or when accessing an address that is actually unused, a load/store access fault exception is raised.

Writing to Read-Only Registers

Unless otherwise specified, writing to registers that are listed as read-only does not trigger an exception as the write access is simply ignored by the corresponding hardware module.

IO Access Latency

In order to shorten the critical path of the IO system, the IO switch provides register stages for the request and response buses.Hence, accesses to the processor-internal IO region require two additional clock cycles to complete.

Module Interrupts

Several peripheral/IO devices provide some kind of interrupt. These interrupts are mapped to the CPU's Custom Fast Interrupt Request Lines. See section Processor Interrupts for more information.

CMSIS System Description View (SVD)

A CMSIS-compatible **System View Description (SVD)** file including all peripherals is available in `sw/svd`.

2.8.1. Instruction Memory (IMEM)

Hardware source files:	neorv32_imem.vhd	default platform-agnostic instruction memory (RAM or ROM)
	neorv32_application_image.vhd	initialization image (a VHDL package)
Software driver files:	none	*implicitly used*
Top entity ports:	none	
Configuration generics:	`MEM_INT_IMEM_EN`	implement processor-internal IMEM when `true`
	`MEM_INT_IMEM_SIZE`	IMEM size in bytes (use a power of 2)
	`BOOT_MODE_SELECT`	implement IMEM as ROM when `BOOT_MODE_SELECT` = 2; see Boot Configuration
CPU interrupts:	none	

Overview

Implementation of the processor-internal instruction memory is enabled by the processor's `MEM_INT_IMEM_EN` generic. The total memory size in bytes is defined via the `MEM_INT_IMEM_SIZE` generic. Note that this size should be a power of two to optimize physical implementation. If enabled, the IMEM is mapped to base address `0x00000000` (see section Address Space).

By default the IMEM is implemented as true RAM so the content can be modified during run time. This is required when using the Bootloader (or the On-Chip Debugger (OCD)) so it can update the content of the IMEM at any time.

Alternatively, the IMEM can be implemented as **pre-initialized read-only memory (ROM)**, so the processor can directly boot from it after reset. This option is configured via the `BOOT_MODE_SELECT` generic. See section Boot Configuration for more information. The initialization image is embedded into the bitstream during synthesis. The software framework provides an option to generate and override the default VHDL initialization file `rtl/core/neorv32_application_image.vhd`, which is automatically inserted into the IMEM (see Makefile Targets. If the IMEM is implemented as RAM (default), the memory block will not be initialized at all.

Platform-Specific Memory Primitives

If required, the default IMEM can be replaced by a platform-/technology-specific primitive to optimize area utilization, timing and power consumption.

Memory Size

If the configured memory size (via the `MEM_INT_IMEM_SIZE` generic) is not a power of two the actual memory size will be auto-adjusted to the next power of two (e.g. configuring a memory size of 60kB will result in a physical memory size of 64kB).

Legacy HDL Style

If synthesis fails to infer block RAM for the IMEM, turn on the `alt_style_c` option inside the memory's VHDL source file. When enabled, a different HDL style is used to describe the memory core.

Read-Only Access

If the IMEM is implemented as ROM any write attempt to it will raise a *store access fault* exception.

2.8.2. Data Memory (DMEM)

Hardware source files:	neorv32_dmem.vhd	default platform-agnostic data memory
Software driver files:	none	*implicitly used*
Top entity ports:	none	
Configuration generics:	`MEM_INT_DMEM_EN`	implement processor-internal DMEM when `true`
	`MEM_INT_DMEM_SIZE`	DMEM size in bytes (use a power of 2)
CPU interrupts:	none	

Overview

Implementation of the processor-internal data memory is enabled by the processor's `MEM_INT_DMEM_EN` generic. The total memory size in bytes is defined via the `MEM_INT_DMEM_SIZE` generic. Note that this size should be a power of two to optimize physical implementation. If the DMEM is implemented, it is mapped to base address `0x80000000` by default (see section Address Space). The DMEM is always implemented as true RAM.

Platform-Specific Memory Primitives

If required, the default DMEM can be replaced by a platform-/technology-specific primitive to optimize area utilization, timing and power consumption.

Memory Size

If the configured memory size (via the `MEM_INT_DMEM_SIZE` generic) is not a power of two the actual memory size will be auto-adjusted to the next power of two (e.g. configuring a memory size of 60kB will result in a physical memory size of 64kB).

Legacy HDL Style

If synthesis fails to infer block RAM for the DMEM, turn on the `alt_style_c` option inside the memory's VHDL source file. When enabled, a different HDL style is used to describe the memory core.

Execute from RAM

The CPU is capable of executing code also from arbitrary data memory.

2.8.3. Bootloader ROM (BOOTROM)

Hardware source files:	neorv32_boot_rom.vhd	default platform-agnostic bootloader ROM
	neorv32_bootloader_image.vhd	initialization image (a VHDL package)
Software driver files:	none	*implicitly used*
Top entity ports:	none	
Configuration generics:	`BOOT_MODE_SELECT`	implement BOOTROM when `BOOT_MODE_SELECT` = 0; see Boot Configuration
CPU interrupts:	none	

Overview

The boot ROM contains the executable image of the default NEORV32 Bootloader. When the Boot Configuration is set to *bootloader* mode (0) via the `BOOT_MODE_SELECT` generic, the boot ROM is automatically enabled and the CPU boot address is adjusted to the base address of the boot ROM. Note that the entire boot ROM is read-only.

Bootloader Image

The bootloader ROM is initialized during synthesis with the default bootloader image (`rtl/core/neorv32_bootloader_image.vhd`). The physical size of the ROM is automatically adjusted to the next power of two of the image size. However, note that the BOOTROM is constrained to a maximum size of 64kB.

2.8.4. Processor-Internal Instruction Cache (iCACHE)

Hardware source files:	neorv32_cache.vhd	Generic cache module
Software driver files:	none	*implicitly used*
Top entity ports:	none	
Configuration generics:	`ICACHE_EN`	implement processor-internal instruction cache when `true`
	`ICACHE_NUM_BLOCKS`	number of cache blocks (pages/lines)
	`ICACHE_BLOCK_SIZE`	size of a cache block in bytes
CPU interrupts:	none	

Overview

The processor features an optional instruction cache to improve performance when using memories with high access latency. The cache is connected directly to the CPU's instruction fetch interface and provides full-transparent accesses. The cache is direct-mapped and read-only.

Cached/Uncached Accesses

The data cache provides direct accesses (= uncached) to memory in order to access memory-mapped IO (like the processor-internal IO/peripheral modules). All accesses that target the address range from `0xF0000000` to `0xFFFFFFFF` will not be cached at all (see section Address Space). Direct/uncached accesses have **lower** priority than cache block operations to allow continuous burst transfer and also to maintain logical instruction forward progress / data coherency. Furthermore, the atomic memory operations of the `Zaamo` ISA Extension will always **bypass** the cache.

Caching Internal Memories

The data cache is intended to accelerate data access to **processor-external** memories. The CPU cache(s) should not be implemented when using only processor-internal data and instruction memories.

Manual Cache Clear/Reload

By executing the `fence.i` instruction the instruction cache is cleared and reloaded. See section Cache Coherency for more information.

Retrieve Cache Configuration from Software

Software can retrieve the cache configuration/layout from the SYSINFO - Cache Configuration register.

Bus Access Fault Handling

The cache always loads a complete cache block (aligned to the block size) every time a cache miss is detected. Each cached word from this block provides a single status bit that indicates if the according bus access was successful or caused a bus error. Hence, the whole cache block remains valid even if certain addresses inside caused a bus error. If the CPU accesses any of the faulty cache words, an instruction bus error exception is raised.

2.8.5. Processor-Internal Data Cache (dCACHE)

Hardware source files:	neorv32_cache.vhd	Generic cache module
Software driver files:	none	*implicitly used*
Top entity ports:	none	
Configuration generics:	`DCACHE_EN`	implement processor-internal data cache when `true`
	`DCACHE_NUM_BLOCKS`	number of cache blocks (pages/lines)
	`DCACHE_BLOCK_SIZE`	size of a cache block in bytes
CPU interrupts:	none	

Overview

The processor features an optional data cache to improve performance when using memories with high access latency. The cache is connected directly to the CPU's data access interface and provides full-transparent accesses. The cache is direct-mapped and uses "write-allocate" and "write-back" strategies.

Cached/Uncached Accesses

The data cache provides direct accesses (= uncached) to memory in order to access memory-mapped IO (like the processor-internal IO/peripheral modules). All accesses that target the address range from `0xF0000000` to `0xFFFFFFFF` will not be cached at all (see section Address Space). Direct/uncached accesses have **lower** priority than cache block operations to allow continuous burst transfer and also to maintain logical instruction forward progress / data coherency. Furthermore, the atomic memory operations of the `Zaamo` ISA Extension will always **bypass** the cache.

Caching Internal Memories

The data cache is intended to accelerate data access to **processor-external** memories. The CPU cache(s) should not be implemented when using only processor-internal data and instruction memories.

Manual Cache Flush/Clear/Reload

By executing the `fence` instruction the data cache is flushed, cleared and reloaded. See section Cache Coherency for more information.

Retrieve Cache Configuration from Software

Software can retrieve the cache configuration/layout from the SYSINFO - Cache Configuration register.

Bus Access Fault Handling

The cache always loads a complete cache block (aligned to the block size) every time a cache miss is detected. Each cached word from this block provides a single status bit that indicates if the according bus access was successful or caused a bus error. Hence, the whole cache block remains valid even if certain addresses inside caused a bus error. If the CPU accesses any of the faulty cache words, a data bus error exception is raised.

2.8.6. Direct Memory Access Controller (DMA)

Hardware source files:	neorv32_dma.vhd	
Software driver files:	neorv32_dma.c	Online software reference (Doxygen)
	neorv32_dma.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	`IO_DMA_EN`	implement DMA when `true`
CPU interrupts:	fast IRQ channel 10	DMA transfer done (see Processor Interrupts)

Overview

The NEORV32 DMA provides a small-scale scatter/gather direct memory access controller that allows to transfer and modify data independently of the CPU. A single read/write transfer channel is implemented that is configured via memory-mapped registers. a configured transfer can either be triggered manually or by a programmable CPU FIRQ interrupt (see NEORV32-Specific Fast Interrupt Requests).

The DMA is connected to the central processor-internal bus system (see section Address Space) and can access the same address space as the CPU core. It uses *interleaving mode* accessing the central processor bus only if the CPU does not currently request and bus access.

The controller can handle different data quantities (e.g. read bytes and write them back as sign-extend words) and can also change the Endianness of data while transferring.

DMA Demo Program

A DMA example program can be found in `sw/example/demo_dma`.

Theory of Operation

The DMA provides four memory-mapped interface registers: A status and control register `CTRL` and three registers for configuring the actual DMA transfer. The base address of the source data is programmed via the `SRC_BASE` register. Vice versa, the base address of the destination data is programmed via the `DST_BASE`. The third configuration register `TTYPE` is use to configure the actual transfer type and the number of elements to transfer.

The DMA is enabled by setting the `DMA_CTRL_EN` bit of the control register. Manual trigger mode (i.e. the DMA transfer is triggered by writing to the `TTYPE` register) is selected if `DMA_CTRL_AUTO` is cleared. Alternatively, the DMA transfer can be triggered by a processor internal FIRQ signal if `DMA_CTRL_AUTO` is set (see section below).

The DMA uses a load-modify-write data transfer process. Data is read from the bus system, internally modified and then written back to the bus system. This combination is implemented as an atomic progress, so canceling the current transfer by clearing the `DMA_CTRL_EN` bit will stop the DMA right after the current load-modify-write operation.

If the DMA controller detects a bus error during operation, it will set either the DMA_CTRL_ERROR_RD (error during last read access) or DMA_CTRL_ERROR_WR (error during last write access) and will terminate the current transfer. Software can read the SRC_BASE or DST_BASE register to retrieve the address that caused the according error. Alternatively, software can read back the NUM bits of the control register to determine the index of the element that caused the error. The error bits are automatically cleared when starting a new transfer.

When the DMA_CTRL_DONE flag is set the DMA has actually executed a transfer. However, the DMA_CTRL_ERROR_* flags should also be checked to verify that the executed transfer completed without errors. The DMA_CTRL_DONE flag is automatically cleared when writing the CTRL register.

DMA Access Privilege Level

Transactions performed by the DMA are executed as bus transactions with elevated **machine-mode** privilege level. Note that any physical memory protection rules (Smpmp ISA Extension) are not applied to DMA transfers.

Transfer Configuration

If the DMA is set to **manual trigger mode** (DMA_CTRL_AUTO = 0) writing the TTRIG register will start the programmed DMA transfer. Once started, the DMA will read one data quantity from the source address, processes it internally and then will write it back to the destination address. The DMA_TTYPE_NUM bits of the TTYPE register define how many times this process is repeated by specifying the number of elements to transfer.

Optionally, the source and/or destination addresses can be increments according to the data quantities automatically by setting the according DMA_TTYPE_SRC_INC and/or DMA_TTYPE_DST_INC bit.

Four different transfer quantities are available, which are configured via the DMA_TTYPE_QSEL bits:

- 00: Read source data as byte, write destination data as byte
- 01: Read source data as byte, write destination data as zero-extended word
- 10: Read source data as byte, write destination data as sign-extended word
- 11: Read source data as word, write destination data as word

Optionally, the DMA controller can automatically convert Endianness of the transferred data if the DMA_TTYPE_ENDIAN bit is set.

Address Alignment

Make sure to align the source and destination base addresses to the according transfer data quantities. For instance, word-to-word transfers require that the two LSB of SRC_BASE and DST_BASE are cleared.

Writing to IO Device

When writing data to IO / peripheral devices (for example to the Cyclic Redundancy Check (CRC)) the destination data quantity has to be set to **word** (32-

bit) since all IO registers can only be written in full 32-bit word mode.

Automatic Trigger

As an alternative to the manual trigger mode, the DMA can be set to **automatic trigger mode** starting a pre-configured transfer if a specific processor-internal peripheral issues a FIRQ interrupt request. The automatic trigger mode is enabled by setting the `CTRL` register's `DMA_CTRL_AUTO` bit. In this configuration *no* transfer is started when writing to the DMA's `TTYPE` register.

The actually triggering FIRQ channel is configured via the control register's `DMA_CTRL_FIRQ_SEL` bits. Writing a 0 will select FIRQ channel 0, writing a 1 will select FIRQ channel 1, and so on. See section Processor Interrupts for a list of all FIRQ channels and their according sources.

The FIRQ trigger can operate in two trigger mode configured via the `DMA_CTRL_FIRQ_TYPE` flag:

- `DMA_CTRL_FIRQ_TYPE = 0`: trigger the automatic DMA transfer on a rising-edge of the selected FIRQ channel (e.g. trigger DMA transfer only once)
- `DMA_CTRL_FIRQ_TYPE = 1`: trigger the automatic DMA transfer when the selected FIRQ channel is active (e.g. trigger DMA transfer again and again)

FIRQ Trigger

The DMA transfer will start if a **rising edge** is detected on the configured FIRQ channel. Hence, the DMA is triggered only once even if the selected FIRQ channel keeps pending.

Memory Barrier / Fence Operation

Optionally, the DMA can issue a FENCE request to the downstream memory system when a transfer has been completed without errors. This can be used to re-sync caches (flush and reload) and buffers to maintain data coherency. This automatic fencing is enabled by the setting the control register's `DMA_CTRL_FENCE` bit.

DMA Interrupt

The DMA features a single CPU interrupt that is triggered when the programmed transfer has completed. This interrupt is also triggered if the DMA encounters a bus error during operation. The interrupt will remain pending until the control register's `DMA_CTRL_DONE` is cleared (this will happen upon any write access to the control register).

Register Map

Table 9. DMA Register Map (`struct NEORV32_DMA`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffed0000`	`CTRL`	`0` `DMA_CTRL_EN`	r/w	DMA module enable
		`1` `DMA_CTRL_AUTO`	r/w	Enable automatic mode (FIRQ-triggered)
		`2` `DMA_CTRL_FENCE`	r/w	Issue a downstream FENCE operation when DMA transfer completes (without errors)
		`7:3` *reserved*	r/-	reserved, read as zero
		`8` `DMA_CTRL_ERROR_RD`	r/-	Error during read access, clears when starting a new transfer
		`9` `DMA_CTRL_ERROR_WR`	r/-	Error during write access, clears when starting a new transfer
		`10` `DMA_CTRL_BUSY`	r/-	DMA transfer in progress
		`11` `DMA_CTRL_DONE`	r/c	Set if a transfer was executed; auto-clears on write-access
		`14:12` *reserved*	r/-	reserved, read as zero
		`15` `DMA_CTRL_FIRQ_TYPE`	r/w	Trigger on rising-edge (`0`) or high-level (`1`) or selected FIRQ channel
		`19:16` `DMA_CTRL_FIRQ_SEL_MSB : DMA_CTRL_FIRQ_SEL_LSB`	r/w	FIRQ trigger select (FIRQ0=0 … FIRQ15=15)
		`31:20` *reserved*	r/-	reserved, read as zero
`0xffed0004`	`SRC_BASE`	`31:0`	r/w	Source base address (shows the last-accessed source address when read)
`0xffed0008`	`DST_BASE`	`31:0`	r/w	Destination base address (shows the last-accessed destination address when read)
`0xffed000c`	`TTYPE`	`23:0` `DMA_TTYPE_NUM_MSB : DMA_TTYPE_NUM_LSB`	r/w	Number of elements to transfer (shows the last-transferred element index when read)
		`26:24` *reserved*	r/-	reserved, read as zero
		`28:27` `DMA_TTYPE_QSEL_MSB : DMA_TTYPE_QSEL_LSB`	r/w	Quantity select (`00` = byte → byte, `01` = byte → zero-extended-word, `10` = byte → sign-extended-word, `11` = word → word)
		`29` `DMA_TTYPE_SRC_INC`	r/w	Constant (`0`) or incrementing (`1`) source address
		`30` `DMA_TTYPE_DST_INC`	r/w	Constant (`0`) or incrementing (`1`) destination address
		`31` `DMA_TTYPE_ENDIAN`	r/w	Swap Endianness when set

2.8.7. Processor-External Bus Interface (XBUS)

Hardware source files:	neorv32_xbus.vhd	External bus gateway
	neorv32_cache.vhd	Generic cache module
Software driver files:	none	*implicitly used*
Top entity ports:	`xbus_adr_o`	address output (32-bit)
	`xbus_dat_o`	data output (32-bit)
	`xbus_tag_o`	access tag (3-bit)
	`xbus_we_o`	write enable (1-bit)
	`xbus_sel_o`	byte enable (4-bit)
	`xbus_stb_o`	bus strobe (1-bit)
	`xbus_cyc_o`	valid cycle (1-bit)
	`xbus_dat_i`	data input (32-bit)
	`xbus_ack_i`	acknowledge (1-bit)
	`xbus_err_i`	bus error (1-bit)
Configuration generics:	`XBUS_EN`	enable external bus interface when `true`
	`XBUS_TIMEOUT`	number of clock cycles after which an unacknowledged external bus access will auto-terminate (0 = disabled)
	`XBUS_REGSTAGE_EN`	implement XBUS register stages
	`XBUS_CACHE_EN`	implement the external bus cache
	`XBUS_CACHE_NUM_BLOCKS`	number of blocks ("lines"), has to be a power of two.
	`XBUS_CACHE_BLOCK_SIZE`	size in bytes of each block, has to be a power of two.
CPU interrupts:	none	

Overview

The external bus interface provides a **Wishbone b4**-compatible on-chip bus interface that is implemented if the `XBUS_EN` generic is `true`. This bus interface can be used to attach processor-external modules like memories, custom hardware accelerators or additional peripheral devices. An optional cache module ("XCACHE") can be enabled to improve memory access latency.

Address Mapping

The external interface is **not** mapped to a specific address space. Instead, all CPU memory accesses that do not target a specific (and actually implemented) processor-internal address region (hence, accessing the "void"; see section Address

Space) are **redirected** to the external bus interface.

AXI4-Lite Interface Bridge

A simple bridge that converts the processor's XBUS into an AXI4-lite-compatible host interface can be found in in `rtl/system_inegration` (`xbus2axi4lite_bridge.vhd`).

AHB3-Lite Interface Bridge

A simple bridge that converts the processor's XBUS into an AHB3-lite-compatible host interface can be found in in `rtl/system_inegration` (`xbus2ahblite_bridge.vhd`).

Wishbone Bus Protocol

The external bus interface complies to the **pipelined Wishbone b4** protocol. Even though this protocol was explicitly designed to support pipelined transfers, only a single transfer will be "in fly" at once. Hence, just two types of bus transactions are generated by the XBUS controller (see images below).

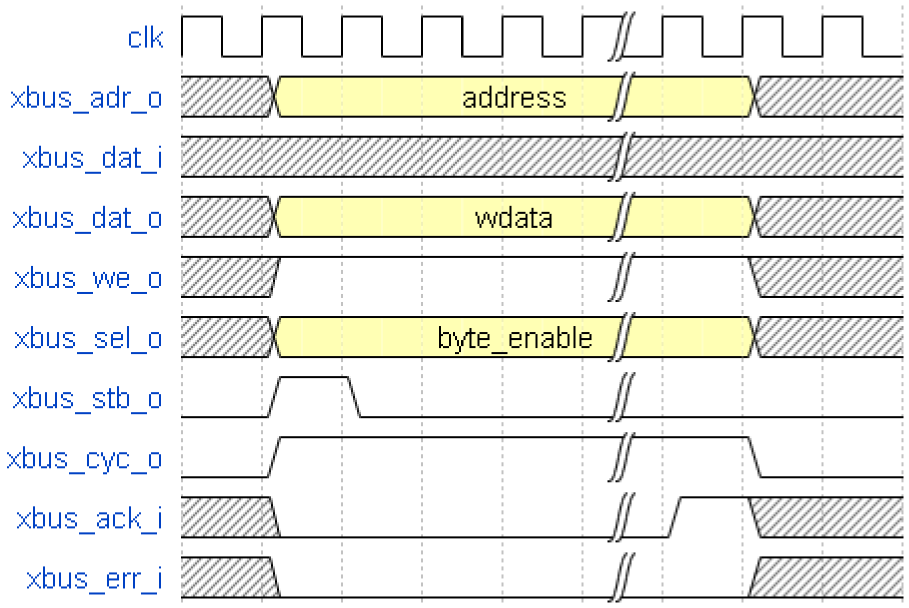


Figure 4. XBUS/Wishbone Write Transaction

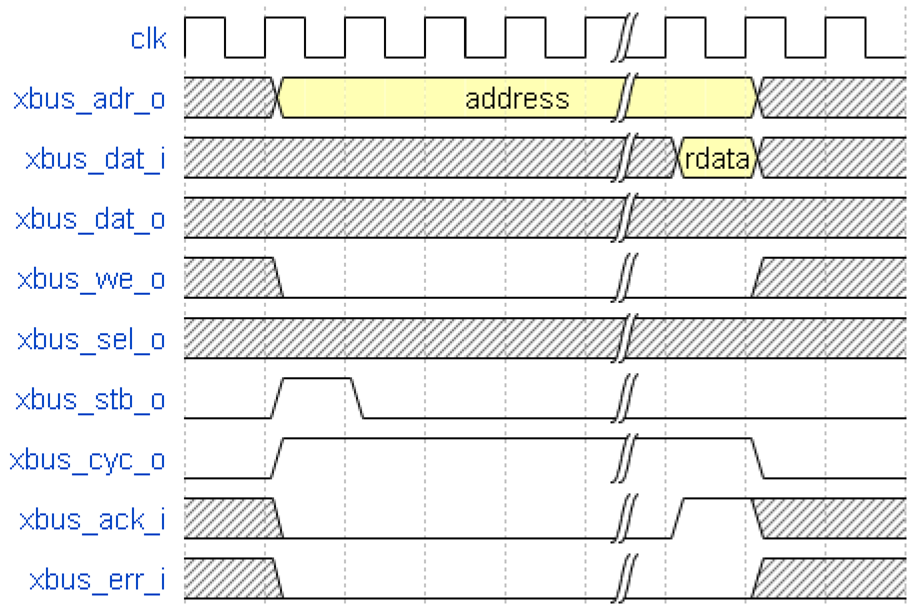


Figure 5. XBUS/Wishbone Read Transaction

Wishbone "Classic" Protocol

Native support for the "classic" Wishbone protocol has been deprecated. However, classic mode can still be *emulated* by connecting the processor's `xbus_cyc_o` directly to the device's / bus system's `cyc` and `stb` signals (omitting the processor's `xbus_stb_o` signal).

Atomic Memory Accesses

[_Atomic_Memory_Access] keep the `cyc` signal active to perform a back-to-back bus access consisting of two `stb` strobes (one for the load/read operation and another one for the store/write operation).

Endianness

Just like the processor itself the XBUS interface uses **little-endian** byte order.

Wishbone Specs.

A detailed description of the implemented Wishbone bus protocol and the according interface signals can be found in the data sheet "Wishbone B4 - WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores". A copy of this document can be found in the `docs` folder of this project.

An accessed XBUS/Wishbone device does not have to respond immediately to a bus request by sending an `ACK`. Instead, there is a **time window** where the device has to acknowledge the transfer. This time window is configured by the `XBUS_TIMEOUT` generic and it defines the maximum time (in clock cycles) a bus access can be pending before it is automatically terminated raising an bus fault exception. If `XBUS_TIMEOUT` is set to zero, the timeout is disabled and a bus access can take an arbitrary number of cycles to complete. Note that this is not recommended as a missing ACK will permanently stall the entire processor!

Furthermore, an accesses XBUS/Wishbone device can signal an error condition at any time by setting the `ERR` signal high for one cycle. This will also terminate the current bus transaction before raising a CPU bus fault exception.

Register Stage

An optional register stage can be added to the XBUS gateway to break up the critical path easing timing closure. When `XBUS_REGSTAGE_EN` is *true* all outgoing and incoming XBUS signals are registered increasing access latency by two cycles. Furthermore, all outgoing signals (like the address) will be kept stable if there is no bus access being initiated.

Access Tag

The XBUS tag signal `xbus_tag_o(0)` provides additional information about the current access cycle. It compatible to the the AXI4 `ARPROT` and `AWPROT` signals.

- `xbus_tag_o(0)` **P**: access is performed from **privileged** mode (machine-mode) when set
- `xbus_tag_o(1)` **NS**: this bit is hardwired to `0` indicating a **secure** access
- `xbus_tag_o(2)` **I**: access is an **instruction** fetch when set; access is a data access when cleared

External Bus Cache (XBUS-CACHE)

The XBUS interface provides an optional internal cache that can be used to buffer processor-external accesses. The x-cache is enabled via the `XBUS_CACHE_EN` generic. The total size of the cache is split into the number of cache lines or cache blocks (`XBUS_CACHE_NUM_BLOCKS` generic) and the line or block size in bytes (`XBUS_CACHE_BLOCK_SIZE` generic).

Listing 6. Simplified X-Cache Architecture

```
                Direct Access        +----------+
          /|------------------------>| Register |------------------------>|\
         | |                         +----------+                         | |
Core --->| |                                                              | |--->
XBUS
         | |    +--------------+    +--------------+    +-------------+    | |
          \|--->| Host Arbiter |--->| Cache Memory |<---| Bus Arbiter |--->|/
                +--------------+    +--------------+    +-------------+
```

The cache uses a direct-mapped architecture that implements "write-allocate" and "write-back" strategies. The **write-allocate** strategy will fetch the entire referenced block from main memory when encountering a cache write-miss. The **write-back** strategy will gather all writes locally inside the cache until the according cache block is about to be replaced. In this case, the entire modified cache block is written back to main memory.

Manual Cache Flush/Clear/Reload

By executing a `fence` or `fence.i` instruction the XBUS cache is flushed (local modifications are send back to main memory), cleared (all cache entries are invalidated) and a reloaded (fetching new data from main memory). See section Cache Coherency for more information.

Cached/Uncached Accesses

The data cache provides direct accesses (= uncached) to memory in order to access memory-mapped IO. All accesses that target the address range from `0xF0000000` to `0xFFFFFFFF` will not be cached at all (see section Address Space). Direct/uncached accesses have **lower** priority than cache block operations to allow continuous burst transfer and also to maintain logical instruction forward progress / data coherency. Furthermore, the atomic memory operations of the `Zaamo` ISA Extension will always **bypass** the cache.

2.8.8. Stream Link Interface (SLINK)

Hardware source files:	neorv32_slink.vhd	
Software driver files:	neorv32_slink.c	Online software reference (Doxygen)
	neorv32_slink.h	Online software reference (Doxygen)
Top entity ports:	`slink_rx_dat_i`	RX link data (32-bit)
	`slink_rx_src_i`	RX routing information (4-bit)
	`slink_rx_val_i`	RX link data valid (1-bit)
	`slink_rx_lst_i`	RX link last element of stream (1-bit)
	`slink_rx_rdy_o`	RX link ready to receive (1-bit)
	`slink_tx_dat_o`	TX link data (32-bit)
	`slink_tx_dst_o`	TX routing information (4-bit)
	`slink_tx_val_o`	TX link data valid (1-bit)
	`slink_tx_lst_o`	TX link last element of stream (1-bit)
	`slink_tx_rdy_i`	TX link allowed to send (1-bit)
Configuration generics:	`IO_SLINK_EN`	implement SLINK when *true*
	`IO_SLINK_RX_FIFO`	RX FIFO depth (1..32k), has to be a power of two, min 1
	`IO_SLINK_TX_FIFO`	TX FIFO depth (1..32k), has to be a power of two, min 1
CPU interrupts:	fast IRQ channel 14	RX SLINK IRQ (see Processor Interrupts)
	fast IRQ channel 15	TX SLINK IRQ (see Processor Interrupts)

Overview

The stream link interface provides independent RX and TX channels for sending and receiving stream data. Each channel features a configurable internal FIFO to buffer stream data (`IO_SLINK_RX_FIFO` for the RX FIFO, `IO_SLINK_TX_FIFO` for the TX FIFO). The SLINK interface provides higher bandwidth and less latency than the external bus interface making it ideally suited for coupling custom stream processors or streaming peripherals.

Example Program

An example program for the SLINK module is available in `sw/example/demo_slink`.

Interface & Protocol

The SLINK interface consists of four signals for each channel:

Copyright by Stephan Nolting. All rights reserved.

- `dat` contains the actual data word
- `val` marks the current transmission cycle as valid
- `lst` marks the current transmission cycle as the last element of a stream
- `rdy` indicates that the receiver is ready to receive
- `src` and `dst` provide source/destination routing information (optional)

AXI4-Stream Compatibility

The interface names (except for `src` and `dst`) and the underlying protocol is compatible to the AXI4-Stream protocol standard. A processor top entity with a AXI4-Stream-compatible interfaces can be found in `rtl/system_inegration`. More information regarding this alternate top entity can be found in in the user guide: https://stnolting.github.io/neorv32/ug/#_packaging_the_processor_as_vivado_ip_block

Theory of Operation

The SLINK provides four interface registers. The control register (`CTRL`) is used to configure the module and to check its status. Two individual data registers (`DATA` and `DATA_LAST`) are used to send and receive the link's actual data stream.

The `DATA` register provides direct access to the RX/TX FIFO buffers. Read accesses return data from the RX FIFO. After reading data from this register the control register's `SLINK_CTRL_RX_LAST` flag can be checked to determine if the according data word has been marked as "end of stream" via the `slink_rx_lst_i` signal (this signal is also buffered by the link's FIFO). Writing to the `DATA` register will immediately write to the TX link FIFO. When writing to the `TX_DATA_LAST` the according data word will also be marked as "end of stream" via the `slink_tx_lst_o` signal (this signal is also buffered by the link's FIFO).

The configured FIFO sizes can be retrieved by software via the control register's `SLINK_CTRL_RX_FIFO_*` and `SLINK_CTRL_TX_FIFO_*` bits.

The SLINK is globally activated by setting the control register's enable bit `SLINK_CTRL_EN`. Clearing this bit will reset all internal logic and will also clear both FIFOs. The FIFOs can also be cleared manually at any time by setting the `SLINK_CTRL_RX_CLR` and/or `SLINK_CTRL_TX_CLR` bits (these bits will auto-clear).

FIFO Overflow

Writing to the TX channel's FIFO while it is *full* will have no effect. Reading from the RX channel's FIFO while it is *empty* will also have no effect and will return the last received data word. There is no overflow indicator implemented yet.

The current status of the RX and TX FIFOs can be determined via the control register's `SLINK_CTRL_RX_*` and `SLINK_CTRL_TX_*` flags.

Stream Routing Information

Both stream link interface provide an optional port for routing information: `slink_tx_dst_o` (AXI stream's `TDEST`) can be used to set a destination address when using a switch/interconnect to access several stream sinks. `slink_rx_src_i` (AXI stream's `TID`) can be used to determine the source when several sources can send data via a switch/interconnect. The routing information can be set/read via the `ROUTE` interface registers. Note that all routing information is also fully buffered by the internal RX/TX FIFOs. RX routing information has to be read **after** reading the according RX data. Vice versa, TX routing information has to be set **before** writing the according TX data.

Interrupts

The SLINK module provides two independent interrupt channels: one for RX events and one for TX events. The interrupt conditions are based on the according link's FIFO status flags and are configured via the control register's `SLINK_CTRL_IRQ_*` flags. The according interrupt will fire when the module is enabled (`SLINK_CTRL_EN`) and the selected interrupt conditions are met. Note that all enabled interrupt conditions are logically OR-ed per channel. If any enable interrupt conditions becomes active the interrupt will become pending until the interrupt-causing condition is resolved (e.g. by reading from the RX FIFO).

Register Map

Table 10. SLINK register map (`struct NEORV32_SLINK`)

Address	Name [C]	Bit(s)	R/W	Function
`0xffec0000`	`CTRL`	`0` `SLINK_CTRL_EN`	r/w	SLINK global enable
		`1` `SLINK_CTRL_RX_CLR`	-/w	Clear RX FIFO when set (bit auto-clears)
		`2` `SLINK_CTRL_TX_CLR`	-/w	Clear TX FIFO when set (bit auto-clears)
		`3` *reserved*	r/-	*reserved*, read as zero
		`4` `SLINK_CTRL_RX_LAST`	r/-	Last word read from `RX_DATA` is marked as "end of stream"
		`7:5` *reserved*	r/-	*reserved*, read as zero
		`8` `SLINK_CTRL_RX_EMPTY`	r/-	RX FIFO empty
		`9` `SLINK_CTRL_RX_HALF`	r/-	RX FIFO at least half full
		`10` `SLINK_CTRL_RX_FULL`	r/-	RX FIFO full
		`11` `SLINK_CTRL_TX_EMPTY`	r/-	TX FIFO empty
		`12` `SLINK_CTRL_TX_HALF`	r/-	TX FIFO at least half full
		`13` `SLINK_CTRL_TX_FULL`	r/-	TX FIFO full
		`15:14` *reserved*	r/-	*reserved*, read as zero
		`16` `SLINK_CTRL_IRQ_RX_NEMPTY`	r/w	RX interrupt if RX FIFO not empty
		`17` `SLINK_CTRL_IRQ_RX_HALF`	r/w	RX interrupt if RX FIFO at least half full
		`18` `SLINK_CTRL_IRQ_RX_FULL`	r/w	RX interrupt if RX FIFO full
		`19` `SLINK_CTRL_IRQ_TX_EMPTY`	r/w	TX interrupt if TX FIFO empty
		`20` `SLINK_CTRL_IRQ_TX_NHALF`	r/w	TX interrupt if TX FIFO not at least half full
		`21` `SLINK_CTRL_IRQ_TX_NFULL`	r/w	TX interrupt if TX FIFO not full
		`23:22` *reserved*	r/-	*reserved*, read as zero
		`27:24` `SLINK_CTRL_RX_FIFO_MSB : SLINK_CTRL_RX_FIFO_LSB`	r/-	log2(RX FIFO size)
		`31:28` `SLINK_CTRL_TX_FIFO_MSB : SLINK_CTRL_TX_FIFO_LSB`	r/-	log2(TX FIFO size)

Address	Name [C]	Bit(s)	R/W	Function
`0xffec0004`	`ROUTE`	`3:0`	r/w	TX destination routing information (`slink_tx_dst_o`)
		`7:4`	r/-	RX source routing information (`slink_rx_src_i`)
		`31:8`	-/-	*reserved*
`0xffec0008`	`DATA`	`31:0`	r/w	Write data to TX FIFO; read data from RX FIFO
`0xffec000c`	`DATA_LAST`	`31:0`	r/w	Write data to TX FIFO (and also set "last" signal); read data from RX FIFO

2.8.9. General Purpose Input and Output Port (GPIO)

Hardware source files:	neorv32_gpio.vhd	
Software driver files:	neorv32_gpio.c	Online software reference (Doxygen)
	neorv32_gpio.h	Online software reference (Doxygen)
Top entity ports:	`gpio_o`	32-bit parallel output port
	`gpio_i`	32-bit parallel input port
Configuration generics:	`IO_GPIO_NUM`	number of input/output pairs to implement (0..32)
CPU interrupts:	fast IRQ channel 8	GPIO (see Processor Interrupts)

Overview

The general purpose IO unit provides simple uni-directional input and output port. These ports can be used chip-externally (for example to drive status LEDs, connect buttons, etc.) or chip-internally to provide control signals for other IP modules. The input port features programmable pin-individual level or edge interrupts capabilities.

Data written to the `PORT_OUT` will appear on the processor's `gpio_o` port. Vice versa, the `PORT_IN` register represents the current state of the processor's `gpio_i`.

The actual number of input/output pairs is defined by the `IO_GPIO_NUM` generic. When set to zero, the GPIO module is excluded from synthesis and the output port `gpio_o` is tied to all-zero. If `IO_GPIO_NUM` is less than the maximum value of 32, only the LSB-aligned bits in `gpio_o` and `gpio_i` are actually connected while the remaining bits are tied to zero or are left unconnected, respectively. This also applies to all memory-mapped interface registers of the GPIO module (i.e. the according most-significant bits are hardwired to zero).

Input Pin Interrupts

Each input pin (`gpio_i`) provides an individual programmable interrupt trigger. The actual interrupt trigger type can be configured individually for each input pin using the `IRQ_TYPE` and `IRQ_POLARITY` registers. `IRQ_TYPE` defines the actual trigger type (level-triggered or edge-triggered), while `IRQ_POLARITY` defines the trigger's polarity (low-level/falling-edge or high-level/rising-edge). The position of each bit in these registers corresponds the according `gpio_i` input pin.

Each pin interrupt channel can be enabled or disabled individually using the `IRQ_ENABLE` register. Each bit in this register corresponds to the according input pin. If the programmed trigger of a disabled input (`IRQ_ENABLE(i) = 0`) fires, the interrupt request is entirely ignored.

Table 11. GPIO Trigger Configuration for Pin i

`IRQ_ENABLE(i)`	`IRQ_TYPE(i)`	`IRQ_POLARITY(i)`	**Resulting trigger of `gpio_i(i)`**
`1`	`0`	`0`	low-level (`GPIO_TRIG_LEVEL_LOW`)
`1`	`0`	`1`	high-level (`GPIO_TRIG_LEVEL_HIGH`)

IRQ_ENABLE(i)	IRQ_TYPE(i)	IRQ_POLARITY(i)	Resulting trigger of gpio_i(i)
1	1	0	falling-edge (GPIO_TRIG_EDGE_FALLING)
1	1	1	rising-edge (GPIO_TRIG_EDGE_RISING)
0	-	-	interrupt disabled

If the configured trigger of an enabled input pin (IRQ_ENABLE(i) = 1) fires, the according interrupt request is buffered internally in the IRQ_PENDING register. When this register contains a non-zero value (i.e. any bit becomes set) an interrupt request is sent to the CPU via FIRQ channel 8 (see Processor Interrupts).

The CPU can determine the interrupt-triggering pins by reading the IRQ_PENDING register. Each set bit in this register indicates that the according input pin's interrupt trigger has fired. Then, the CPU can clear those pending interrupt pin by setting all set bits to zero.

GPIO Interrupts Demo Program

A demo program for the GPIO input interrupts can be found in sw/example/demo_gpio.

Register Map

Table 12. GPIO unit register map (struct NEORV32_GPIO)

Address	Name [C]	Bit(s)	R/W	Function
0xfffc0000	PORT_IN	31:0	r/-	Parallel input port; PORT_IN(i) corresponds to gpio_i(i)
0xfffc0004	PORT_OUT	31:0	r/w	Parallel output port; PORT_OUT(i) corresponds to gpio_o(i)
0xfffc0008	-	31:0	r/-	*reserved*, read as zero
0xfffc000c	-	31:0	r/-	*reserved*, read as zero
0xfffc0010	IRQ_TYPE	31:0	r/w	Trigger type select (0 = level trigger, 1 = edge trigger); IRQ_TYPE(i) corresponds to gpio_i(i)
0xfffc0014	IRQ_POLARITY	31:0	r/w	Trigger polarity select (0 = low-level/falling-edge, 1 = high-level/rising-edge); IRQ_POLARITY(i) corresponds to gpio_i(i)
0xfffc0018	IRQ_ENABLE	31:0	r/w	Per-pin interrupt enable; IRQ_ENABLE(i) corresponds to gpio_i(i)
0xfffc001c	IRQ_PENDING	31:0	r/c	Per-pin interrupt pending, can be cleared by writing zero to the according bit(s); IRQ_PENDING(i) corresponds to gpio_i(i)

2.8.10. Cyclic Redundancy Check (CRC)

Hardware source files:	neorv32_crc.vhd	
Software driver files:	neorv32_crc.c	Online software reference (Doxygen)
	neorv32_crc.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	`IO_CRC_EN`	implement CRC module when `true`
CPU interrupts:	none	

Overview

The cyclic redundancy check unit provides a programmable checksum computation module. The unit operates on single bytes and can either compute CRC8, CRC16 or CRC32 checksums based on an arbitrary polynomial and start value.

CRC Demo Program

A CRC example program (also using CPU-independent DMA transfers) can be found in `sw/example/crc_dma`.

CPU-Independent Operation

The CRC unit can compute a checksum for an arbitrary memory array without any CPU overhead by using the processor's Direct Memory Access Controller (DMA).

Theory of Operation

The module provides four interface registers:

- `MODE`: selects either CRC8-, CRC16- or CRC32-mode
- `POLY`: programmable polynomial
- `DATA`: data input register (single bytes only)
- `SREG`: the CRC shift register; this register is used to define the start value and to obtain the final processing result

The `MODE`, `POLY` and `SREG` registers need to be programmed before the actual processing can be started. Writing a byte to `DATA` will update the current checksum in `SREG`.

Access Latency

Write access to the CRC module have an increased latency of 8 clock cycles. This additional latency ensures that the internal bit-serial processing of the current data byte has also been completed when the transfer is completed.

Data Size

For CRC8-mode only bits `7:0` of `POLY` and `SREG` are relevant; for CRC16-mode only bits `15:0` are used and for CRC32-mode the entire 32-bit of `POLY` and `SREG` are used.

Register Map

Table 13. CRC Register Map (`struct NEORV32_CRC`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffee0000`	`CTRL`	`1:0`	r/w	CRC mode select (`00` CRC8, `01`: CRC16, `10`: CRC32)
		`31:2`	r/-	*reserved*, read as zero
`0xffee0004`	`POLY`	`31:0`	r/w	CRC polynomial
`0xffee0008`	`DATA`	`7:0`	r/w	data input (single byte)
		`31:8`	r/-	*reserved*, read as zero, writes are ignored
`0xffee000c`	`SREG`	`32:0`	r/w	current CRC shift register value (set start value on write)

2.8.11. Watchdog Timer (WDT)

Hardware source files:	neorv32_wdt.vhd	
Software driver files:	neorv32_wdt.c	Online software reference (Doxygen)
	neorv32_wdt.h	Online software reference (Doxygen)
Top entity ports:	`rstn_wdt_o`	synchronous watchdog reset output, low-active
Configuration generics:	`IO_WDT_EN`	implement watchdog when `true`
CPU interrupts:	none	

Overview

The watchdog (WDT) provides a last resort for safety-critical applications. When a pre-programmed timeout value is reached a system-wide hardware reset is generated. The internal counter has to be reset explicitly by the application program every now and then to prevent a timeout.

Theory of Operation

The watchdog is enabled by setting the control register's `WDT_CTRL_EN` bit. When this bit is cleared, the internal timeout counter is reset to zero and no system reset can be triggered by this module.

The internal 32-bit timeout counter is clocked at 1/4096th of the processor's main clock (f_{WDT}[Hz] = f_{main}[Hz] / 4096). Whenever this counter reaches the programmed timeout value (`WDT_CTRL_TIMEOUT` bits in the control register) a hardware reset is triggered.

The watchdog's timeout counter is reset ("feeding the watchdog") by writing the reset **PASSWORD** to the `RESET` register. The password is hardwired to hexadecimal `0x709D1AB3`.

Watchdog Operation during Debugging

By default, the watchdog stops operation when the CPU enters debug mode and will resume normal operation after the CPU has left debug mode again. This will prevent an unintended watchdog timeout during a debug session. However, the watchdog can also be configured to keep operating even when the CPU is in debug mode by setting the control register's `WDT_CTRL_DBEN` bit.

Watchdog Operation during CPU Sleep

By default, the watchdog stops operating when the CPU enters sleep mode. However, the watchdog can also be configured to keep operating even when the CPU is in sleep mode by setting the control register's `WDT_CTRL_SEN` bit.

Configuration Lock

The watchdog control register can be *locked* to protect the current configuration from being modified. The lock is activated by setting the `WDT_CTRL_LOCK` bit. In the locked state any write access to the control register is entirely ignored (see table below, "writable if locked"). However, read accesses to the control register as well as watchdog resets are further possible.

The lock bit can only be set if the WDT is already enabled (`WDT_CTRL_EN` is set). Furthermore, the lock bit can only be cleared again by a system-wide hardware reset.

Strict Mode

The *strict operation mode* provides additional safety functions. If the strict mode is enabled by the `WDT_CTRL_STRICT` control register bit an **immediate hardware** reset if enforced if

- the `RESET` register is written with an incorrect password or
- the `CTRL` register is written and the `WDT_CTRL_LOCK` bit is set.

Cause of last Hardware Reset

The cause of the last system hardware reset can be determined via the `WDT_CTRL_RCAUSE_*` bits:

- `WDT_RCAUSE_EXT` (0b00): Reset caused by external reset signal/pin
- `WDT_RCAUSE_OCD` (0b01): Reset caused by on-chip debugger
- `WDT_RCAUSE_TMO` (0b10): Reset caused by watchdog timeout
- `WDT_RCAUSE_ACC` (0b11): Reset caused by illegal watchdog access (strict mode)

External Reset Output

The WDT provides a dedicated output (Processor Top Entity - Signals: `rstn_wdt_o`) to reset processor-external modules when the watchdog times out. This signal is low-active and synchronous to the processor clock. It is available if the watchdog is implemented; otherwise it is hardwired to `1`. Note that the signal also becomes active (low) when the processor's main reset signal is active (even if the watchdog is deactivated or disabled for synthesis).

Register Map

Table 14. WDT register map (`struct NEORV32_WDT`)

Address	Name [C]	Bit(s), Name [C]	R/W	Reset value	Writable if locked	Function
`0xfffb0000`	`CTRL`	`0` `WDT_CTRL_EN`	r/w	`0`	no	watchdog enable
		`1` `WDT_CTRL_LOCK`	r/w	`0`	no	lock configuration when set, clears only on system reset, can only be set if enable bit is set already
		`2` `WDT_CTRL_DBEN`	r/w	`0`	no	set to allow WDT to continue operation even when CPU is in debug mode
		`3` `WDT_CTRL_SEN`	r/w	`0`	no	set to allow WDT to continue operation even when CPU is in sleep mode
		`4` `WDT_CTRL_STRICT`	r/w	`0`	no	set to enable strict mode (force hardware reset if reset password is incorrect or if write access to locked CTRL register)
		`6:5` `WDT_CTRL_RCAUSE_HI : WDT_CTRL_RCAUSE_LO`	r/-	`0`	-	cause of last system reset; 0=external reset, 1=ocd-reset, 2=watchdog reset
		`7` -	r/-	-	-	*reserved*, reads as zero
		`31:8` `WDT_CTRL_TIMEOUT_MSB : WDT_CTRL_TIMEOUT_LSB`	r/w	0	no	24-bit watchdog timeout value
`0xfffb0004`	`RESET`	`31:0`	-/w	-	yes	Write *PASSWORD* to reset WDT timeout counter

2.8.12. Core Local Interruptor (CLINT)

Hardware source files:	neorv32_clint.vhd	
Software driver files:	neorv32_clint.c	Online software reference (Doxygen)
	neorv32_clint.h	Online software reference (Doxygen)
Top entity ports:	`mtime_irq_i`	RISC-V machine timer IRQ if CLINT is **not** implemented
	`msw_irq_i`	RISC-V software IRQ if CLINT is **not** implemented
	`mtime_time_o`	Current system time (from CLINT's MTIMER)
Configuration generics:	`IO_CLINT_EN`	implement core local interruptor when `true`
CPU interrupts:	`MTI`	machine timer interrupt (see Processor Interrupts)
	`MSI`	machine software interrupt (see Processor Interrupts)

Overview

The core local interruptor provides machine-level timer and software interrupts for a set of CPU cores (also called *harts). It is compatible to the original SiFive® CLINT specifications and it is also backwards-compatible to the upcoming RISC-V _Advanced Core Local Interruptor (ACLINT)* specifications. In terms of the ACLINT spec the NEORV32 CLINT implements three *devices* that are placed next to each other in the address space: an MTIMER and an MSWI device.

The CLINT can support up to 4095 harts. However, the NEORV32 CLINT is configured for a single hart only (yet). Hence, only the according registers are implemented while the remaining ones are hardwired to zero.

MTIMER Device

The MTIMER device provides a global 64-bit machine timer (`NEORV32_CLINT→MTIME`) that increments with every main processor clock tick. Upon reset the timer is reset to all zero. Each hart provides an individual 64-bit timer-compare register (`NEORV32_CLINT→MTIMECMP[0]` for hart 0). Whenever `MTIMECMP >= MTIME` the according machine timer interrupt is pending.

MSIW Device

The MSIV provides software interrupts for each hart via hart-individual memory-mapped registers (`NEORV32_CLINT→MSWI[0]` for hart 0). Setting bit 0 of this register will bring the machine software interrupt into pending state.

External Machine Timer and Software Interrupts

If the NEORV32 CLINT module is disabled (`IO_CLINT_EN` = `false`) the core's machine timer interrupt and machine software interrupt become available as processor-external signals (`mtime_irq_i` and `msw_irq_i`, respectively).

Register Map

Table 15. CLINT register map (`struct NEORV32_CLINT`)

Address	Name [C]	Bits	R/W	Function
`0xfff40000`	`MSWI[0]`	0	r/w	trigger machine software interrupt for hart 0 when set
		31:1	r/-	hardwired to zero
`0xfff40004`	`MSWI[1]`	0	r/w	trigger machine software interrupt for hart 1 when set
		31:1	r/-	hardwired to zero
`0xfff44000`	`MTIMECMP[0]`	63:0	r/w	64-bit time compare for hart 0
`0xfff44008`	`MTIMECMP[1]`	63:0	r/w	64-bit time compare for hart 1
`0xfff4bff8`	`MTIME`	63:0	r/w	64-bit global machine timer

2.8.13. Primary Universal Asynchronous Receiver and Transmitter (UART0)

Hardware source files:	neorv32_uart.vhd	
Software driver files:	neorv32_uart.c	Online software reference (Doxygen)
	neorv32_uart.h	Online software reference (Doxygen)
Top entity ports:	`uart0_txd_o`	serial transmitter output
	`uart0_rxd_i`	serial receiver input
	`uart0_rts_o`	flow control: RX ready to receive, low-active
	`uart0_cts_i`	flow control: RX ready to receive, low-active
Configuration generics:	`IO_UART0_EN`	implement UART0 when `true`
	`UART0_RX_FIFO`	RX FIFO depth (power of 2, min 1)
	`UART0_TX_FIFO`	TX FIFO depth (power of 2, min 1)
CPU interrupts:	fast IRQ channel 2	RX interrupt
	fast IRQ channel 3	TX interrupt (see Processor Interrupts)

Overview

The NEORV32 UART provides a standard serial interface with independent transmitter and receiver channels, each equipped with a configurable FIFO. The transmission frame is fixed to **8N1**: 8 data bits, no parity bit, 1 stop bit. The actual transmission rate (Baud rate) is programmable via software. The module features two memory-mapped registers: `CTRL` and `DATA`. These are used for configuration, status check and data transfer.

Standard Console

All default example programs and software libraries of the NEORV32 software framework (including the bootloader and the runtime environment) use the primary UART (*UART0*) as default user console interface. Furthermore, UART0 is used to implement the "standard consoles" (`STDIN`, `STDOUT` and `STDERR`).

RX and TX FIFOs

The UART provides individual data FIFOs for RX and TX to allow data transmission without CPU intervention. The sizes of these FIFOs can be configured via the according configuration generics (`UART0_RX_FIFO` and `UART0_TX_FIFO`). Both FIFOs a re automatically cleared when disabling the module via the `UART_CTRL_EN` flag. However, the FIFOs can also be cleared individually by setting the `UART_CTRL_RX_CLR` / `UART_CTRL_TX_CLR` flags.

Theory of Operation

The module is enabled by setting the `UART_CTRL_EN` bit in the UART0 control register `CTRL`. The Baud rate is configured via a 10-bit UART_CTRL_BAUDx baud divisor (baud_div) and a 3-bit UART_CTRL_PRSCx clock prescaler (clock_prescaler).

Table 16. UART0 Clock Configuration

UART_CTRL_PRSCx	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting clock_prescaler	2	4	8	64	128	1024	2048	4096

Baud rate = (f_{main}*[Hz]* / clock_prescaler) / (baud_div + 1)

The control register's UART_CTRL_RX_* and UART_CTRL_TX_* flags provide information about the RX and TX FIFO fill level. Disabling the module via the UART_CTRL_EN bit will also clear these FIFOs.

A new TX transmission is started by writing to the DATA register. The transfer is completed when the UART_CTRL_TX_BUSY control register flag returns to zero. RX data is available when the UART_CTRL_RX_NEMPTY flag becomes set. The UART_CTRL_RX_OVER will be set if the RX FIFO overflows. This flag is cleared only by disabling the module via UART_CTRL_EN.

UART Interrupts

The UART module provides independent interrupt channels for RX and TX. These interrupts are triggered by certain RX and TX FIFO levels. The actual configuration is programmed independently for the RX and TX interrupt channel via the control register's UART_CTRL_IRQ_RX_* and UART_CTRL_IRQ_TX_* bits:

1. **RX IRQ** The RX interrupt can be triggered by three different RX FIFO level states: If UART_CTRL_IRQ_RX_NEMPTY is set the interrupt fires if the RX FIFO is *not* empty (e.g. when incoming data is available). If UART_CTRL_IRQ_RX_HALF is set the RX IRQ fires if the RX FIFO is at least half-full. If UART_CTRL_IRQ_RX_FULL the interrupt fires if the RX FIFO is full. Note that all these programmable conditions are logically OR-ed (interrupt fires if any enabled conditions is true).
2. **TX IRQ** The TX interrupt can be triggered by two different TX FIFO level states: If UART_CTRL_IRQ_TX_EMPTY is set the interrupt fires if the TX FIFO is empty. If UART_CTRL_IRQ_TX_NHALF is set the interrupt fires if the TX FIFO is *not* at least half full. Note that all these programmable conditions are logically OR-ed (interrupt fires if any enabled conditions is true).

Once an UART interrupt has fired it remains pending until the actual cause of the interrupt is resolved; for example if just the UART_CTRL_IRQ_RX_NEMPTY bit is set, the RX interrupt will keep firing until the RX FIFO is empty again.

RX/TX FIFO Size

Software can retrieve the configured sizes of the RX and TX FIFO via the according UART_DATA_RX_FIFO_SIZE and UART_DATA_TX_FIFO_SIZE bits from the DATA register.

RTS/CTS Hardware Flow Control

The NEORV32 UART supports optional hardware flow control using the standard CTS uart0_cts_i ("clear to send") and RTS uart0_rts_o ("ready to send" / "ready to receive (RTR)") signals. Both signals are low-active. Hardware flow control is enabled by setting the `UART_CTRL_HWFC_EN` bit in the modules control register `CTRL`.

When hardware flow control is enabled:

1. The UART's transmitter will not start a new transmission until the `uart0_cts_i` signal goes low. During this time, the UART busy flag `UART_CTRL_TX_BUSY` remains set.
2. The UART will set `uart0_rts_o` signal low if the RX FIFO is **less than half full** (to have a wide safety margin). As long as this signal is low, the connected device can send new data. `uart0_rts_o` is always low if the hardware flow-control is disabled. Disabling the UART (setting `UART_CTRL_EN` low) while having hardware flow-control enabled, will set `uart0_rts_o` high to signal that the UARt is not capable of receiving new data.

Note that RTS and CTS signaling can only be activated together. If the RTS handshake is not required the signal can be left unconnected. If the CTS handshake is not required it has to be tied to zero.

Simulation Mode

The UART provides a *simulation-only* mode to dump console data as well as raw data directly to a file. When the simulation mode is enabled (by setting the `UART_CTRL_SIM_MODE` bit) there will be **no** physical transaction on the `uart0_txd_o` signal. Instead, all data written to the `DATA` register is immediately dumped to a file. Data written to `DATA[7:0]` will be dumped as ASCII chars to a file named `neorv32.uart0_sim_mode.out`. Additionally, the ASCII data is printed to the simulator console.

Both file are created in the simulation's home folder.

Register Map

Table 17. UART0 register map (`struct NEORV32_UART0`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff50000`	`CTRL`	`0` `UART_CTRL_EN`	r/w	UART enable
		`1` `UART_CTRL_SIM_MODE`	r/w	enable **simulation mode**
		`2` `UART_CTRL_HWFC_EN`	r/w	enable RTS/CTS hardware flow-control
		`5:3` `UART_CTRL_PRSC2 : UART_CTRL_PRSC0`	r/w	Baud rate clock prescaler select
		`15:6` `UART_CTRL_BAUD9 : UART_CTRL_BAUD0`	r/w	12-bit Baud value configuration value
		`16` `UART_CTRL_RX_NEMPTY`	r/-	RX FIFO not empty
		`17` `UART_CTRL_RX_HALF`	r/-	RX FIFO at least half-full
		`18` `UART_CTRL_RX_FULL`	r/-	RX FIFO full
		`19` `UART_CTRL_TX_EMPTY`	r/-	TX FIFO empty
		`20` `UART_CTRL_TX_NHALF`	r/-	TX FIFO not at least half-full
		`21` `UART_CTRL_TX_FULL`	r/-	TX FIFO full
		`22` `UART_CTRL_IRQ_RX_NEMPTY`	r/w	fire IRQ if RX FIFO not empty
		`23` `UART_CTRL_IRQ_RX_HALF`	r/w	fire IRQ if RX FIFO at least half-full
		`24` `UART_CTRL_IRQ_RX_FULL`	r/w	fire IRQ if RX FIFO full
		`25` `UART_CTRL_IRQ_TX_EMPTY`	r/w	fire IRQ if TX FIFO empty
		`26` `UART_CTRL_IRQ_TX_NHALF`	r/w	fire IRQ if TX not at least half full
		`27` -	r/-	*reserved* read as zero
		`28` `UART_CTRL_RX_CLR`	r/w	Clear RX FIFO, flag auto-clears
		`29` `UART_CTRL_TX_CLR`	r/w	Clear TX FIFO, flag auto-clears
		`30` `UART_CTRL_RX_OVER`	r/-	RX FIFO overflow; cleared by disabling the module
		`31` `UART_CTRL_TX_BUSY`	r/-	TX busy or TX FIFO not empty

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff50004`	`DATA`	`7:0 UART_DATA_RTX_MSB : UART_DATA_RTX_LSB`	r/w	receive/transmit data
		`11:8 UART_DATA_RX_FIFO_SIZE_MSB : UART_DATA_RX_FIFO_SIZE_LSB`	r/-	log2(RX FIFO size)
		`15:12 UART_DATA_TX_FIFO_SIZE_MSB : UART_DATA_TX_FIFO_SIZE_LSB`	r/-	log2(TX FIFO size)
		`31:16`	r/-	*reserved*, read as zero

2.8.14. Secondary Universal Asynchronous Receiver and Transmitter (UART1)

Hardware source files:	neorv32_uart.vhd	
Software driver files:	neorv32_uart.c	
	neorv32_uart.h	
Top entity ports:	`uart1_txd_o`	serial transmitter output
	`uart1_rxd_i`	serial receiver input
	`uart1_rts_o`	flow control: RX ready to receive, low-active
	`uart1_cts_i`	flow control: RX ready to receive, low-active
Configuration generics:	`IO_UART1_EN`	implement UART1 when `true`
	`UART1_RX_FIFO`	RX FIFO depth (power of 2, min 1)
	`UART1_TX_FIFO`	TX FIFO depth (power of 2, min 1)
CPU interrupts:	fast IRQ channel 4	RX interrupt
	fast IRQ channel 5	TX interrupt (see Processor Interrupts)
Access restrictions:	privileged access only, non-32-bit write accesses are ignored	

Overview

The secondary UART (UART1) is functionally identical to the primary UART (Primary Universal Asynchronous Receiver and Transmitter (UART0)). Obviously, UART1 uses different addresses for the control register (`CTRL`) and the data register (`DATA`). The register's bits/flags use the same bit positions and naming as for the primary UART. The RX and TX interrupts of UART1 are mapped to different CPU fast interrupt (FIRQ) channels.

Simulation Mode

The secondary UART (UART1) provides the same simulation options as the primary UART (UART0). However, output data is written to UART1-specific file `neorv32.uart1_sim_mode.out`. This data is also printed to the simulator console.

Register Map

Table 18. UART1 register map (`struct NEORV32_UART1`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff60000`	`CTRL`	…	…	Same as UART0
`0xfff60004`	`DATA`	…	…	Same as UART0

2.8.15. Serial Peripheral Interface Controller (SPI)

Hardware source files:	neorv32_spi.vhd	
Software driver files:	neorv32_spi.c	Online software reference (Doxygen)
	neorv32_spi.h	Online software reference (Doxygen)
Top entity ports:	`spi_clk_o`	1-bit serial clock output
	`spi_dat_o`	1-bit serial data output
	`spi_dat_i`	1-bit serial data input
	`spi_csn_o`	8-bit dedicated chip select output (low-active)
Configuration generics:	`IO_SPI_EN`	implement SPI controller when `true`
	`IO_SPI_FIFO`	FIFO depth, has to be a power of two, min 1
CPU interrupts:	fast IRQ channel 6	configurable SPI interrupt (see Processor Interrupts)

Overview

The NEORV32 SPI module is a **host** transceiver. Hence, it is responsible for generating transmission. The module operates on a byte.wide data granularity, supports all 4 standard clock modes, a fine-tunable SPI clock generator and provides up to 8 dedicated chip select signals via the top entity's `spi_csn_o` signal. An optional receive/transmit ring-buffer/FIFO can be configured via the `IO_SPI_FIFO` generic to support block-based transmissions without CPU interaction.

Host-Mode Only

The NEORV32 SPI module only supports *host mode*. Transmission are initiated only by the processor's SPI module and not by an external SPI module. If you are looking for a *device-mode* serial peripheral interface (transactions initiated by an external host) check out the Serial Data Interface Controller (SDI).

The SPI module provides a single control register `CTRL` to configure the module and to check it's status and a single data register `DATA` for receiving/transmitting data.

Theory of Operation

The SPI module is enabled by setting the `SPI_CTRL_EN` bit in the `CTRL` control register. No transfer can be initiated and no interrupt request will be triggered if this bit is cleared. Clearing this bit will reset the entire module, clear the FIFO and terminate any transfer being in process.

The actual SPI transfer (receiving one byte while sending one byte) as well as control of the chip-select lines is handled via the module's `DATA` register. Note that this register will access the TX FIFO of the ring-buffer when writing and will access the RX FIFO of the ring-buffer when reading.

The most significant bit of the `DATA` register (`SPI_DATA_CMD`) is used to select the purpose of the data being written. When the `SPI_DATA_CMD` is cleared, the lowest 8-bit represent the actual SPI TX data. This data will be transmitted by the SPI bus engine. After completion, the received data is stored to the RX FIFO.

If `SPI_DATA_CMD` is cleared, the lowest 4-bit control the chip-select lines. In this case, bis `2:0` select one of the eight chip-select lines. The selected line will become enabled when bit `3` is also set. If bit `3` is cleared, all chip-select lines will be disabled.

Examples:

- Enable chip-select line 3: `NEORV32_SPI->DATA = (1 << SPI_DATA_CMD) | (1 << 3) | 3;`
- Enable chip-select line 7: `NEORV32_SPI->DATA = (1 << SPI_DATA_CMD) | (1 << 3) | 7;`
- Disable all chip-select lines: `NEORV32_SPI->DATA = (1 << SPI_DATA_CMD) | (0 << 3);`
- Send data byte `0xAB`: `NEORV32_SPI->DATA = (0 << SPI_DATA_CMD) | 0xAB;`

Since all SPI operations are controlled via the FIFO, entire SPI sequences (chip-enable, data transmissions, chip-disable) can be "programmed". Thus, SPI operations can be executed without any CPU interaction at all.

Application software can check if any chip-select is enabled by reading the control register's `SPI_CS_ACTIVE` flag.

SPI Clock Configuration

The SPI module supports all standard SPI clock modes (0, 1, 2, 3), which are configured via the two control register bits `SPI_CTRL_CPHA` and `SPI_CTRL_CPOL`. The `SPI_CTRL_CPHA` bit defines the *clock phase* and the `SPI_CTRL_CPOL` bit defines the *clock polarity*.

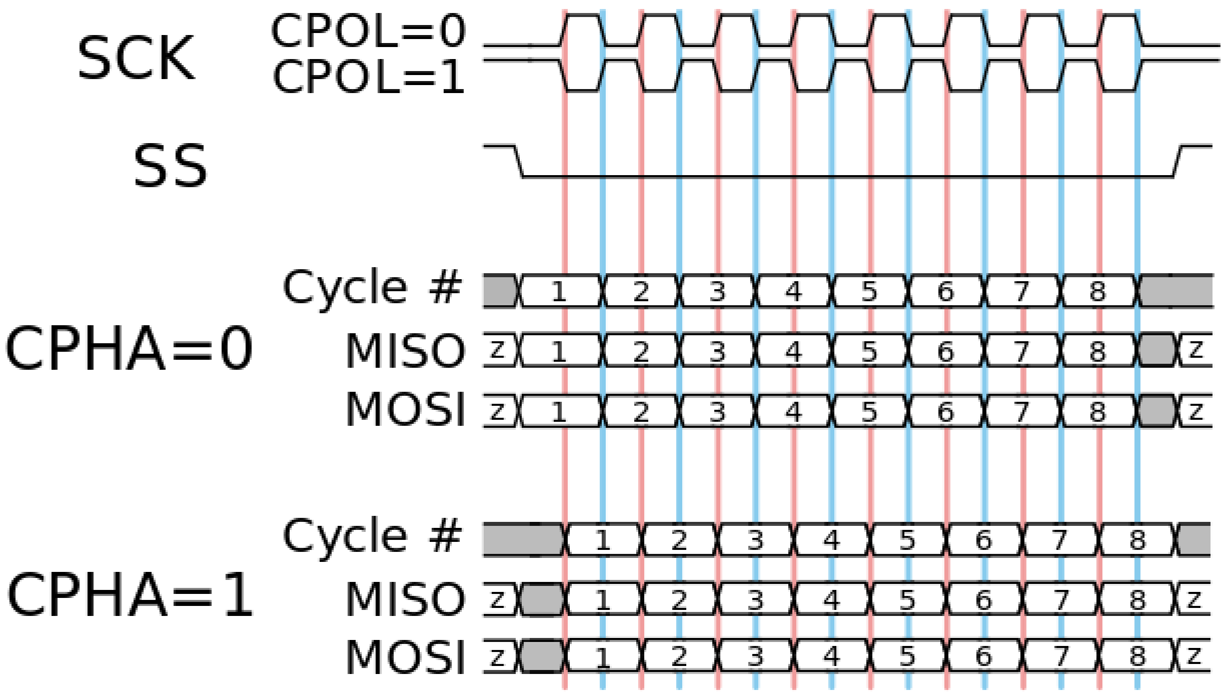


Figure 6. SPI clock modes; image from https://en.wikipedia.org/wiki/File:SPI_timing_diagram2.svg (license: (Wikimedia) Creative Commons Attribution-Share Alike 3.0 Unported)

The SPI clock frequency (`spi_clk_o`) is programmed by the 3-bit `SPI_CTRL_PRSCx` clock prescaler for a coarse clock selection and a 4-bit clock divider `SPI_CTRL_CDIVx` for a fine clock configuration. The following clock prescalers (`SPI_CTRL_PRSCx`) are available:

Table 19. SPI prescaler configuration

`SPI_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

Based on the programmed clock configuration, the actual SPI clock frequency f_{SPI} is derived from the processor's main clock f_{main} according to the following equation:

$f_{SPI} = f_{main}[Hz] / (2 * \texttt{clock\_prescaler} * (1 + \texttt{SPI\_CTRL\_CDIVx}))$

Hence, the maximum SPI clock is f_{main} / 4 and the lowest SPI clock is f_{main} / 131072. The SPI clock is always symmetric having a duty cycle of exactly 50%.

High-Speed Mode

The SPI provides a high-speed mode to further boost the maximum SPI clock frequency. When enabled via the control register's `SPI_CTRL_HIGHSPEED` bit the clock prescaler configuration (`SPI_CTRL_PRSCx` bits) is overridden setting it to a minimal factor of 1. However, the clock speed can still be fine-tuned using the `SPI_CTRL_CDIVx` bits.

$f_{SPI} = f_{main}[Hz] / (2 * 1 * (1 + \texttt{SPI\_CTRL\_CDIVx}))$

Hence, the maximum SPI clock is f_{main} / 2 when in high-speed mode.

SPI Interrupt

The SPI module provides a set of programmable interrupt conditions based on the level of the RX/TX FIFO. The different interrupt sources are enabled by setting the according control register's `SPI_CTRL_IRQ_*` bits. All enabled interrupt conditions are logically OR-ed, so any enabled interrupt source will trigger the module's interrupt signal.

Once the SPI interrupt has fired it remains pending until the actual cause of the interrupt is resolved; for example if just the `SPI_CTRL_IRQ_RX_AVAIL` bit is set, the interrupt will keep firing until the RX FIFO is empty again.

Register Map

Table 20. SPI register map (`struct NEORV32_SPI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff80000`	`CTRL`	`0` `SPI_CTRL_EN`	r/w	SPI module enable
		`1` `SPI_CTRL_CPHA`	r/w	clock phase
		`2` `SPI_CTRL_CPOL`	r/w	clock polarity
		`5:3` `SPI_CTRL_PRSC2 : SPI_CTRL_PRSC0`	r/w	3-bit clock prescaler select
		`9:6` `SPI_CTRL_CDIV2 : SPI_CTRL_CDIV0`	r/w	4-bit clock divider for fine-tuning
		`10` `SPI_CTRL_HIGHSPEED`	r/w	high-speed mode enable (overriding `SPI_CTRL_PRSC*`)
		`15:11` *reserved*	r/-	reserved, read as zero
		`16` `SPI_CTRL_RX_AVAIL`	r/-	RX FIFO data available (RX FIFO not empty)
		`17` `SPI_CTRL_TX_EMPTY`	r/-	TX FIFO empty
		`18` `SPI_CTRL_TX_NHALF`	r/-	TX FIFO *not* at least half full
		`19` `SPI_CTRL_TX_FULL`	r/-	TX FIFO full
		`20` `SPI_CTRL_IRQ_RX_AVAIL`	r/w	Trigger IRQ if RX FIFO not empty
		`21` `SPI_CTRL_IRQ_TX_EMPTY`	r/w	Trigger IRQ if TX FIFO empty
		`22` `SPI_CTRL_IRQ_TX_NHALF`	r/w	Trigger IRQ if TX FIFO *not* at least half full
		`23` `SPI_CTRL_IRQ_IDLE`	r/w	Trigger IRQ if TX FIFO is empty and SPI bus engine is idle
		`27:24` `SPI_CTRL_FIFO_MSB : SPI_CTRL_FIFO_LSB`	r/-	FIFO depth; log2(`IO_SPI_FIFO`)
		`30:28` *reserved*	r/-	reserved, read as zero
		`30` `SPI_CS_ACTIVE`	r/-	Set if any chip-select line is active
		`31` `SPI_CTRL_BUSY`	r/-	SPI module busy when set (serial engine operation in progress and TX FIFO not empty yet)
`0xfff80004`	`DATA`	`7:0` `SPI_DATA_MSB : SPI_DATA_LSB`	r/w	receive/transmit data (FIFO)
		`30:8` *reserved*	r/-	reserved, read as zero
		`31` `SPI_DATA_CMD`	-/w	data (`0`) / chip-select-command (`1`) select

2.8.16. Serial Data Interface Controller (SDI)

Hardware source files:	neorv32_sdi.vhd	
Software driver files:	neorv32_sdi.c	Online software reference (Doxygen)
	neorv32_sdi.h	Online software reference (Doxygen)
Top entity ports:	`sdi_clk_i`	1-bit serial clock input
	`sdi_dat_o`	1-bit serial data output
	`sdi_dat_i`	1-bit serial data input
	`sdi_csn_i`	1-bit chip-select input (low-active)
Configuration generics:	`IO_SDI_EN`	implement SDI controller when `true`
	`IO_SDI_FIFO`	data FIFO size, has to a power of two, min 1
CPU interrupts:	fast IRQ channel 11	configurable SDI interrupt (see Processor Interrupts)

Overview

The serial data interface module provides a **device-class** SPI interface and allows to connect the processor to an **external SPI host**, which is responsible of performing the actual transmission - the SDI is entirely passive. An optional receive/transmit ring buffer (FIFOs) can be configured via the `IO_SDI_FIFO` generic to support block-based transmissions without CPU interaction.

Device-Mode Only

The NEORV32 SDI module only supports *device mode*. Transmission are initiated by an external host and not by the the processor itself. If you are looking for a *host-mode* serial peripheral interface (transactions performed by the NEORV32) check out the Serial Peripheral Interface Controller (SPI).

The SDI module provides a single control register `CTRL` to configure the module and to check it's status and a single data register `DATA` for receiving/transmitting data. Any access to the `DATA` register actually accesses the internal ring buffer.

Theory of Operation

The SDI module is enabled by setting the `SDI_CTRL_EN` bit in the `CTRL` control register. Clearing this bit resets the entire module and will also clear the entire RX/TX ring buffer.

The SDI operates on byte-level only. Data written to the `DATA` register will be pushed to the TX FIFO. Received data can be retrieved by reading the RX FIFO via the `DATA` register. The current state of these FIFOs is available via the control register's `SDI_CTRL_RX_*` and `SDI_CTRL_TX_*` flags. If no data is available in the TX FIFO while an external device performs a transmission the external device will read all-zero from the SDI controller.

Application software can check the current state of the SDI chip-select input via the control register's `SDI_CTRL_CS_ACTIVE` flag (the flag is set when the chip-select line is active (pulled low)).

MSB-first Only

The NEORV32 SDI module only supports MSB-first mode.

In-Transmission Abort

If the external SPI controller aborts the transmission by setting the chip-select signal high again *before* 8 data bits have been transferred, no data is written to the RX FIFO.

SDI Clocking

The SDI module supports both SPI clock polarity modes ("CPOL") but only "CPHA=0"-clock-phase is *officially* supported yet. However, experiments have shown that the SDI module can also deal with both clock phase modes (for slow SDI clock speeds).

All SDI operations are clocked by the external `sdi_clk_i` signal. This signal is synchronized to the processor's clock domain to simplify timing behavior. This clock synchronization requires the external SDI clock (`sdi_clk_i`) does **not exceed 1/4 of the processor's main clock**.

SDI Interrupt

The SDI module provides a set of programmable interrupt conditions based on the level of the RX & TX FIFOs. The different interrupt sources are enabled by setting the according control register's `SDI_CTRL_IRQ_*` bits. All enabled interrupt conditions are logically OR-ed so any enabled interrupt source will trigger the module's interrupt signal.

Once the SDI interrupt has fired it will remain active until the actual cause of the interrupt is resolved; for example if just the `SDI_CTRL_IRQ_RX_AVAIL` bit is set, the interrupt will keep firing until the RX FIFO is empty again.

Register Map

Table 21. SDI register map (`struct NEORV32_SDI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff70000`	`CTRL`	`0` `SDI_CTRL_EN`	r/w	SDI module enable
		`3:1` *reserved*	r/-	reserved, read as zero
		`7:4` `SDI_CTRL_FIFO_MSB : SDI_CTRL_FIFO_LSB`	r/-	FIFO depth; log2(*IO_SDI_FIFO*)
		`14:8` *reserved*	r/-	reserved, read as zero
		`15` `SDI_CTRL_IRQ_RX_AVAIL`	r/w	fire interrupt if RX FIFO is not empty
		`16` `SDI_CTRL_IRQ_RX_HALF`	r/w	fire interrupt if RX FIFO is at least half full
		`17` `SDI_CTRL_IRQ_RX_FULL`	r/w	fire interrupt if if RX FIFO is full
		`18` `SDI_CTRL_IRQ_TX_EMPTY`	r/w	fire interrupt if TX FIFO is empty
		`19` `SDI_CTRL_IRQ_TX_NHALF`	r/w	fire interrupt if TX FIFO is not at least half full
		`22:20` *reserved*	r/-	reserved, read as zero
		`23` `SDI_CTRL_RX_AVAIL`	r/-	RX FIFO data available (RX FIFO not empty)
		`24` `SDI_CTRL_RX_HALF`	r/-	RX FIFO at least half full
		`25` `SDI_CTRL_RX_FULL`	r/-	RX FIFO full
		`26` `SDI_CTRL_TX_EMPTY`	r/-	TX FIFO empty
		`27` `SDI_CTRL_TX_NHALF`	r/-	TX FIFO not at least half full
		`28` `SDI_CTRL_TX_FULL`	r/-	TX FIFO full
		`30:29` *reserved*	r/-	reserved, read as zero
		`31` `SDI_CTRL_CS_ACTIVE`	r/-	Chip-select is active when set
`0xfff70004`	`DATA`	`7:0`	r/w	receive/transmit data (FIFO)
		`31:8` *reserved*	r/-	reserved, read as zero

2.8.17. Two-Wire Serial Interface Controller (TWI)

Hardware source files:	neorv32_twi.vhd	
Software driver files:	neorv32_twi.c	Online software reference (Doxygen)
	neorv32_twi.h	Online software reference (Doxygen)
Top entity ports:	`twi_sda_i`	1-bit serial data line sense input
	`twi_sda_o`	1-bit serial data line output (pull low only)
	`twi_scl_i`	1-bit serial clock line sense input
	`twi_scl_o`	1-bit serial clock line output (pull low only)
Configuration generics:	`IO_TWI_EN`	implement TWI controller when `true`
	`IO_TWI_FIFO`	FIFO depth, has to be a power of two, min 1
CPU interrupts:	fast IRQ channel 7	FIFO empty and module idle interrupt (see Processor Interrupts)

Overview

The NEORV32 TWI implements a I²C-compatible host controller to communicate with arbitrary I2C-devices. Note that peripheral-mode (controller acts as a device) and multi-controller mode are not supported yet.

Host-Mode Only

The NEORV32 TWI controller only supports **host mode**. Transmission are initiated by the processor's TWI controller and not by an external I²C module. If you are looking for a *device-mode* module (transactions initiated by an external host) check out the Two-Wire Serial Device Controller (TWD).

Key features:

- Programmable clock speed
- Optional clock stretching
- Generate START / repeated-START and STOP conditions
- Sending & receiving 8 data bits including ACK/NACK
- Generating a host-ACK (ACK send by the TWI controller)
- Configurable data/command FIFO to "program" large I²C sequences without further involvement of the CPU

The TWI controller provides two memory-mapped registers that are used for configuring the module and for triggering operations: the control and status register `CTRL` and the command and data register DCMD.

Tristate Drivers

The TWI module requires two tristate drivers (actually: open-drain drivers - signals can only be actively driven low) for the SDA and SCL lines, which have to be implemented by the user in the setup's top module / IO ring. A generic VHDL example is shown below (here, sda_io and scl_io are the actual I²C bus lines, which are of type std_logic).

Listing 7. TWI VHDL Tristate Driver Example

```
sda_io    <= '0' when (twi_sda_o = '0') else 'Z'; -- drive
scl_io    <= '0' when (twi_scl_o = '0') else 'Z'; -- drive
twi_sda_i <= std_ulogic(sda_io); -- sense
twi_scl_i <= std_ulogic(scl_io); -- sense
```

TWI Clock Speed

The TWI clock frequency is programmed by two bit-fields in the device's control register CTRL: a 3-bit clock prescaler (TWI_CTRL_PRSCx) is used for a coarse clock configuration and a 4-bit clock divider (TWI_CTRL_CDIVx) is used for a fine clock configuration.

Table 22. TWI prescaler configuration

TWI_CTRL_PRSCx	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting clock_prescaler	2	4	8	64	128	1024	2048	4096

Based on the clock configuration, the actual TWI clock frequency f_{SCL} is derived from the processor's main clock f_{main} according to the following equation:

$f_{SCL} = f_{main}[Hz] / (4 * \texttt{clock\_prescaler} * (1 + \text{TWI\_CTRL\_CDIV}))$

Hence, the maximum TWI clock is f_{main} / 8 and the lowest TWI clock is f_{main} / 262144. The generated TWI clock is always symmetric having a duty cycle of exactly 50% (if the clock is not haled by a device during clock stretching).

Clock Stretching

An accessed peripheral can slow down/halt the controller's bus clock by using clock stretching (= actively keeping the SCL line low). The controller will halt operation in this case. Clock stretching is enabled by setting the TWI_CTRL_CLKSTR bit in the module's control register CTRL.

TWI Transfers

The TWI is enabled via the TWI_CTRL_EN bit in the CTRL control register. All TWI operations are controlled by the DCMD register. The actual operation is selected by a 2-bit value that is written to the register's TWI_DCMD_CMD_* bit-field:

- `00`: NOP (no-operation); all further bit-fields in `DCMD` are ignored
- `01`: Generate a (repeated) START conditions; all further bit-fields in `DCMD` are ignored
- `10`: Generate a STOP conditions; all further bit-fields in `DCMD` are ignored
- `11`: Trigger a data transmission; the data to be send has to be written to the register's `TWI_DCMD_MSB : TWI_DCMD_LSB` bit-field; if `TWI_DCMD_ACK` is set the controller will send a host-ACK in the ACK/NACK time slot; after the transmission is completed `TWI_DCMD_MSB : TWI_DCMD_LSB` contains the RX data and `TWI_DCMD_ACK` the device's response if no host-ACK was configured (`0` = ACK, `1` = ACK)

All operations/data written to the `DCMD` register are buffered by a configurable data/command FIFO. The depth of the FIFO is configured by the `IO_TWI_FIFO` top generic. Software can retrieve this size by reading the control register's `TWI_CTRL_FIFO` bits.

The command/data FIFO is internally split into a TX FIFO and a RX FIFO. Writing to `DCMD` will write to the TX FIFO while reading from `DCMD` will read from the RX FIFO. The TX FIFO is full when the `TWI_CTRL_TX_FULL` flag is set. Accordingly, the RX FIFO contains valid data when the `TWI_CTRL_RX_AVAIL` flag is set.

The control register's busy flag `TWI_CTRL_BUSY` is set as long as the TX FIFO contains valid data (i.e. programmed TWI operations that have not been executed yet) or of the TWI bus engine is still processing an operation.

An active transmission can be terminated at any time by disabling the TWI module. This will also clear the data/command FIFO.

The current state of the I²C bus lines (SCL and SDA) can be checked by software via the `TWI_CTRL_SENSE_*` control register bits.

When reading data from a device, an all-one byte (`0xFF`) has to be written to TWI data register `NEORV32_TWI.DATA` so the accessed device can actively pull-down SDA when required.

TWI Interrupt

The TWI module provides a single interrupt to signal "idle condition" to the CPU. The interrupt becomes active when the TWI module is enabled (`TWI_CTRL_EN` = `1`) and the TX FIFO is empty and the TWI bus engine is idle.

Register Map

Table 23. TWI register map (`struct NEORV32_TWI`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff90000`	`CTRL`	`0` `TWI_CTRL_EN`	r/w	TWI enable, reset if cleared
		`3:1` `TWI_CTRL_PRSC2 : TWI_CTRL_PRSC0`	r/w	3-bit clock prescaler select
		`7:4` `TWI_CTRL_CDIV3 : TWI_CTRL_CDIV0`	r/w	4-bit clock divider
		`8` `TWI_CTRL_CLKSTR`	r/w	Enable (allow) clock stretching
		`14:9` -	r/-	*reserved*, read as zero
		`18:15` `TWI_CTRL_FIFO_MSB : TWI_CTRL_FIFO_LSB`	r/-	FIFO depth; log2(`IO_TWI_FIFO`)
		`26:12` -	r/-	*reserved*, read as zero
		`27` `TWI_CTRL_SENSE_SCL`	r/-	current state of the SCL bus line
		`28` `TWI_CTRL_SENSE_SDA`	r/-	current state of the SDA bus line
		`29` `TWI_CTRL_TX_FULL`	r/-	set if the TWI bus is claimed by any controller
		`30` `TWI_CTRL_RX_AVAIL`	r/-	RX FIFO data available
		`31` `TWI_CTRL_BUSY`	r/-	TWI bus engine busy or TX FIFO not empty
`0xfff90004`	`DCMD`	`7:0` `TWI_DCMD_MSB : TWI_DCMD_LSB`	r/w	RX/TX data byte
		`8` `TWI_DCMD_ACK`	r/w	write: ACK bit sent by controller; read: `1` = device NACK, `0` = device ACK
		`10:9` `TWI_DCMD_CMD_HI : TWI_DCMD_CMD_LO`	r/w	TWI operation (`00` = NOP, `01` = START conditions, `10` = STOP condition, `11` = data transmission)

2.8.18. Two-Wire Serial Device Controller (TWD)

Hardware source files:	neorv32_twd.vhd	
Software driver files:	neorv32_twd.c	Online software reference (Doxygen)
	neorv32_twd.h	Online software reference (Doxygen)
Top entity ports:	`twd_sda_i`	1-bit serial data line sense input
	`twd_sda_o`	1-bit serial data line output (pull low only)
	`twd_scl_i`	1-bit serial clock line sense input
	`twd_scl_o`	1-bit serial clock line output (pull low only)
Configuration generics:	`IO_TWD_EN`	implement TWD controller when `true`
	`IO_TWD_FIFO`	RX/TX FIFO depth, has to be a power of two, min 1
CPU interrupts:	fast IRQ channel 0	FIFO status interrupt (see Processor Interrupts)

Overview

The NEORV32 TWD implements a I2C-compatible **device-mode** controller. Processor-external hosts can communicate with this module by issuing I2C transactions. The TWD is entirely passive an only reacts on those external transmissions.

Key features:

- Programmable 7-bit device address
- Programmable interrupt conditions
- Configurable RX/TX data FIFO to "program" large TWD sequences without further involvement of the CPU

Device-Mode Only

The NEORV32 TWD controller only supports **device mode**. Transmission are initiated by processor-external modules and not by an external TWD. If you are looking for a *host-mode* module (transactions initiated by the processor) check out the Two-Wire Serial Interface Controller (TWI).

Theory of Operation

The TWD module provides two memory-mapped registers that are used for configuration & status check (`CTRL`) and for accessing transmission data (`DATA`). The `DATA` register is transparently buffered by separate RX and TX FIFOs. The size of those FIFOs can be configured by the `IO_TWD_FIFO` generic. Software can determine the FIFO size via the control register's `TWD_CTRL_FIFO_*` bits.

The module is globally enabled by setting the control register's `TWD_CTRL_EN` bit. Clearing this bit will disable and reset the entire module also clearing the internal RX and TX FIFOs. Each FIFO can also be cleared individually at any time by setting `TWD_CTRL_CLR_RX` or `TWD_CTRL_CLR_TX`, respectively.

The external two wire bus is sampled sampled and synchronized into the processor's clock domain with a sampling frequency of 1/8 of the processor's main clock. In order to increase the resistance to glitches the sampling frequency can be lowered to 1/64 of the processor clock by setting the control register's `TWD_CTRL_FSEL` bit.

Current Bus State

The current state of the I2C bus lines (SCL and SDA) can be checked by software via the `TWD_CTRL_SENSE_*` control register bits. Note that the TWD module needs to be enabled in order to sample the bus state.

The actual 7-bit device address of the TWD is programmed by the `TWD_CTRL_DEV_ADDR` bits. Note that the TWD will only response to a host transactions if the host issues the according address. Specific general-call or broadcast addresses are not supported.

Depending on the transaction type, data is either read from the RX FIFO and transferred to the host ("read operation") or data is received from the host and written to the TX FIFO ("write operation"). Hence, data sequences can be programmed to the TX FIFO to be fetched from the host. If the TX FIFO is empty and the host keeps performing read transaction, the transferred data byte is automatically set to all-one.

The current status of the RX and TX FIFO can be polled by software via the `TWD_CTRL_RX_*` and `TWD_CTRL_TX_*` flags.

TWD Interrupt

The TWD module provides a single interrupt to signal certain FIFO conditions to the CPU. The control register's `TWD_CTRL_IRQ_*` bits are used to enabled individual interrupt conditions. Note that all enabled conditions are logically OR-ed.

- `TWD_CTRL_IRQ_RX_AVAIL`: trigger interrupt if at least one data byte is available in the RX FIFO
- `TWD_CTRL_IRQ_RX_FULL`: trigger interrupt if the RX FIFO is completely full
- `TWD_CTRL_IRQ_TX_EMPTY`: trigger interrupt if the TX FIFO is empty

The interrupt remains active until all enabled interrupt-causing conditions are resolved. The interrupt can only trigger if the module is actually enabled (`TWD_CTRL_EN` is set).

TWD Transmissions

Two standard I2C-compatible transaction types are supported: **read** operations and **write** operations. These two operation types are illustrated in the following figure (note that the transactions are split across two lines to improve readability).

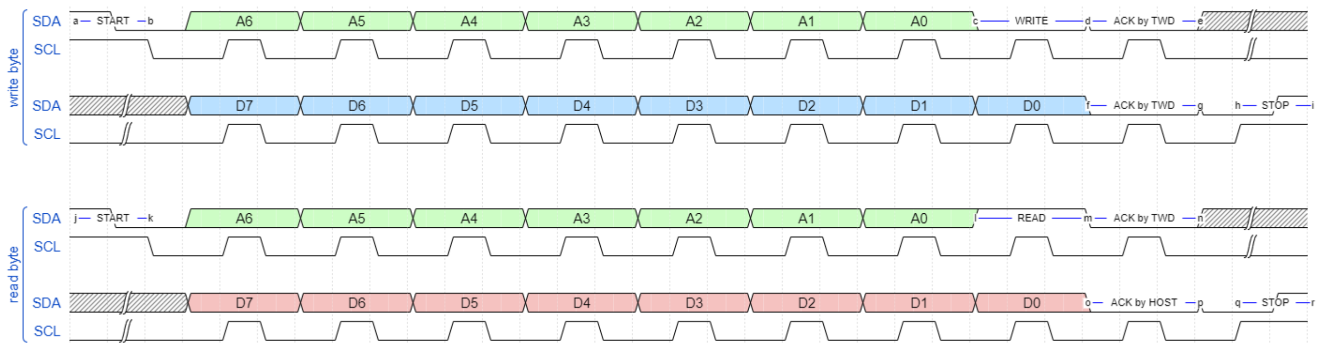


Figure 7. TWD single-byte read and write transaction timing (not to scale)

Any new transaction starts with a **START** condition. Then, the host transmits the 7 bit device address MSB-first (green signals `A6` to `A0`) plus a command bit. The command bit can be either **write** (pulling the SDA line low) or **read** (leaving the SDA line high). If the transferred address matches the one programmed to to `TWD_CTRL_DEV_ADDR` control register bits the TWD module will response with an **ACK** (acknowledge) by pulling the SDA bus line actively low during the 9th SCL clock pulse. If there is no address match the TWD will not interfere with the bus and move back to idle state.

For a **write transaction** (upper timing diagram) the host can now transfer an arbitrary number of bytes (blue signals `D7` to `D0`, MSB-first) to the TWD module. Each byte is acknowledged by the TWD by pulling SDA low during the 9th SCL clock pules (**ACK**). Each received data byte is pushed to the internal RX FIFO. Data will be lost if the FIFO overflows. The transaction is terminated when the host issues a **STOP** condition after the TWD has acknowledged the last data transfer.

For a **read transaction** (lower timing diagram) the host keeps the SDA line at high state while sending the clock pulse. The TWD will read a byte from the internal TX FIFO and will transmit it MSB-first to the host (blue signals `D7` to `D0)`. During the 9th clock pulse the host has to acknowledged the transfer (**ACK**) by pulling SDA low. If no ACK is received by the TWD no data is taken from the TX FIFO and the same byte can be transmitted in the next data phase. If the TX FIFO becomes empty while the host keeps reading data, all-one bytes are transmitted. To terminate the transmission the host hast so send a **NACK** after receiving the last data byte by keeping SDA high. After that, the host has to issue a **STOP** condition.

A **repeated-START** condition can be issued at any time (but after the complete transaction of a data byte and there according ACK/NACK) bringing the TWD back to the start of the address/command transmission phase. The control register's `TWD_CTRL_BUSY` flag remains high while a bus transaction is in progress.

Abort / Termination

An active or even stuck transmission can be terminated at any time by disabling the TWD module. This will also clear the RX/TX FIFOs.

Tristate Drivers

The TWD module requires two tristate drivers (actually: open-drain drivers - signals can only be actively driven low) for the SDA and SCL lines, which have to be implemented by the user in the setup's top module / IO ring. A generic VHDL example is shown below (here, `sda_io` and `scl_io` are the actual TWD bus lines, which are of type `std_logic`).

Listing 8. TWD VHDL Tristate Driver Example

```
sda_io    <= '0' when (twd_sda_o = '0') else 'Z'; -- drive
scl_io    <= '0' when (twd_scl_o = '0') else 'Z'; -- drive
twd_sda_i <= std_ulogic(sda_io); -- sense
twd_scl_i <= std_ulogic(scl_io); -- sense
```

Register Map

Table 24. TWD register map (`struct NEORV32_TWD`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffea0000`	`CTRL`	`0` `TWD_CTRL_EN`	r/w	TWD enable, reset if cleared
		`1` `TWD_CTRL_CLR_RX`	-/w	Clear RX FIFO, flag auto-clears
		`2` `TWD_CTRL_CLR_TX`	-/w	Clear TX FIFO, flag auto-clears
		`3` `TWD_CTRL_FSEL`	r/w	Bus sample clock / filter select
		`10:4` `TWD_CTRL_DEV_ADDR6 : TWD_CTRL_DEV_ADDR0`	r/w	Device address (7-bit)
		`11` `TWD_CTRL_IRQ_RX_AVAIL`	r/w	IRQ if RX FIFO data available
		`12` `TWD_CTRL_IRQ_RX_FULL`	r/w	IRQ if RX FIFO full
		`13` `TWD_CTRL_IRQ_TX_EMPTY`	r/w	IRQ if TX FIFO empty
		`14:9` -	r/-	*reserved*, read as zero
		`18:15` `TWD_CTRL_FIFO_MSB : TWD_CTRL_FIFO_LSB`	r/-	FIFO depth; log2(`IO_TWD_FIFO`)
		`24:12` -	r/-	*reserved*, read as zero
		`25` `TWD_CTRL_RX_AVAIL`	r/-	RX FIFO data available
		`26` `TWD_CTRL_RX_FULL`	r/-	RX FIFO full
		`27` `TWD_CTRL_TX_EMPTY`	r/-	TX FIFO empty
		`28` `TWD_CTRL_TX_FULL`	r/-	TX FIFO full
		`29` `TWD_CTRL_SENSE_SCL`	r/-	current state of the SCL bus line
		`30` `TWD_CTRL_SENSE_SDA`	r/-	current state of the SDA bus line
		`31` `TWD_CTRL_BUSY`	r/-	bus engine is busy (transaction in progress)
`0xffea0004`	`DATA`	`7:0` `TWD_DATA_MSB : TWD_DATA_LSB`	r/w	RX/TX data FIFO access
		`31:8` -	r/-	*reserved*, read as zero

2.8.19. One-Wire Serial Interface Controller (ONEWIRE)

Hardware source files:	neorv32_onewire.vhd	
Software driver files:	neorv32_onewire.c	Online software reference (Doxygen)
	neorv32_onewire.h	Online software reference (Doxygen)
Software reference:	Online Doxygen	
Top entity ports:	`onewire_i`	1-bit 1-wire bus sense input
	`onewire_o`	1-bit 1-wire bus output (pull low only)
Configuration generics:	`IO_ONEWIRE_EN`	implement ONEWIRE interface controller when `true`
	`IO_ONEWIRE_FIFO`	RTX fifo depth, has to be zero or a power of two, min 1
CPU interrupts:	fast IRQ channel 13	operation done interrupt (see Processor Interrupts)

Overview

The NEORV32 ONEWIRE module implements a single-wire interface controller that is compatible to the Dallas/Maxim 1-Wire protocol, which is an asynchronous half-duplex bus requiring only a single signal wire (plus ground) for communication.

The bus is based on a single open-drain signal. The controller as well as all devices on the bus can only pull-down the bus (similar to TWI/I2C). The default high-level is provided by a single pull-up resistor connected to the positive power supply close to the bus controller. Recommended values are between 1kΩ and 10kΩ depending on the bus characteristics (wire length, number of devices, etc.).

Tri-State Drivers

The ONEWIRE module requires a tristate driver (actually, just an open-drain driver) for the 1-wire bus line, which has to be implemented in the top module / IO ring of the design. A generic VHDL example is given below (`onewire_io` is the actual 1-wire bus signal, which is of type `std_logic`; `onewire_o` and `onewire_i` are the processor's ONEWIRE port signals).

Listing 9. ONEWIRE VHDL Tristate Driver Example

```
onewire_io   <= '0' when (onewire_o = '0') else 'Z'; -- drive (low)
onewire_i <= std_ulogic(onewire_io); -- sense
```

Theory of Operation

The ONEWIRE controller provides two interface registers: `CTRL` and `DCMD`. The control register (`CTRL`) is used to configure the module and to monitor the current state. The `DCMD` register, which can optionally by buffered by a configurable FIFO (`IO_ONEWIRE_FIFO` generic), is used to read/write data from/to the bus and to trigger bus operations.

The module is enabled by setting the `ONEWIRE_CTRL_EN` bit in the control register. If this bit is cleared, the module is automatically reset, any bus operation is aborted, the bus is brought to high-level (due to the external pull-up resistor) and the internal FIFO is cleared. The basic timing configuration is programmed via a coarse clock prescaler (`ONEWIRE_CTRL_PRSCx` bits) and a fine clock divider (`ONEWIRE_CTRL_CLKDIVx` bits).

The controller can execute four basic bus operations, which are triggered by writing the according command bits in the `DCMD` register (`ONEWIRE_DCMD_DATA_*` bits) while also writing the actual data bits (`ONEWIRE_DCMD_CMD_*` bits).

1. `0b00` (`ONEWIRE_CMD_NOP`) - no operation (dummy)
2. `0b01` (`ONEWIRE_CMD_BIT`) - transfer a single-bit (read-while-write)
3. `0b10` (`ONEWIRE_CMD_BYTE`) - transfer a full-byte (read-while-write)
4. `0b11` (`ONEWIRE_CMD_RESET`) - generate reset pulse and check for device presence

Every command (except NOP) will result in a bus operation when dispatched from the data/command FIFO. Each command (except NOP) will also sample a bus response (a read bit, a read byte or a presence pulse) to a shadowed receive FIFO that is accessed when reading the `DCMD` register.

When the single-bit operation (`ONEWIRE_CMD_BIT`) is executed, the data previously written to `DCMD[0]` will be send to the bus and the response is sample to `DCMD[7]`. Accordingly, a full-byte transmission (`ONEWIRE_CMD_BYTE`) will send the byte written to `DCMD[7:0]` to the bus and will sample the response to `DCMD[7:0]` (LSB-first). Finally, the reset command (`ONEWIRE_CMD_RESET`) will generate a bus reset and will also sample the "presence pulse" from the device(s) to the `DCMD[ONEWIRE_DCMD_PRESENCE]`.

Read from Bus

In order to read a single bit from the bus `DCMD[0]` has to set to `1` before triggering the bit transmission operation to allow the accessed device to pull-down the bus. Accordingly, `DCMD[7:0]` has to be set to `0xFF` before triggering the byte transmission operation when the controller shall read a byte from the bus.

As soon as the current bus operation has completed (and there are no further operations pending in the FIFO) the `ONEWIRE_CTRL_BUSY` bit in the control registers clears.

Bus Timing

The control register provides a 2-bit clock prescaler select (`ONEWIRE_CTRL_PRSC`) and a 8-bit clock divider (`ONEWIRE_CTRL_CLKDIV`) for timing configuration. Both are used to define the elementary base time T_{base}. All bus operations are timed using multiples of this elementary base time.

Table 25. ONEWIRE Clock Prescaler Configurations

`ONEWIRE_CTRL_PRSCx`	`0b00`	`0b01`	`0b10`	`0b11`
Resulting `clock_prescaler`	2	4	8	64

Together with the clock divider value (`ONEWIRE_CTRL_PRSCx` bits = `clock_divider`) the base time is defined by the following formula:

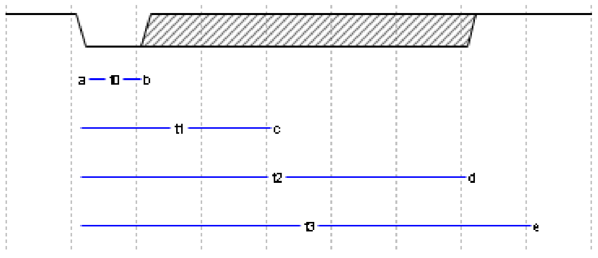
T_{base} = (1 / $f_{main}[Hz]$) * `clock_prescaler` * (`clock_divider` + 1)

Example:

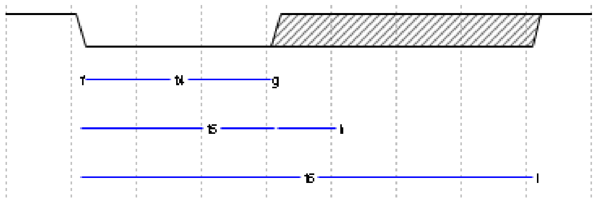
- f_{main} = 100MHz
- clock prescaler select = `0b01` → `clock_prescaler` = 4
- clock divider `clock_divider` = 249

T_{base} = (1 / 100000000Hz) * 4 * (249 + 1) = 10000ns = **10µs**

The base time is used to coordinate all bus interactions. Hence, all delays, time slots and points in time are quantized as multiples of the base time T_{base}. The following images show the two basic operations of the ONEWIRE controller: single-bit (0 or 1) transaction and reset with presence detect. Note that the full-byte operations just repeats the single-bit operation eight times. The relevant points in time are shown as *absolute* time points (in multiples of the time base T_{base}) with the falling edge of the bus as reference points.



Single-bit data transmission (not to scale)



Reset pulse and presence detect (not to scale)

Table 26. Data Transmission Timing

Symbol	Description	Multiples of T_{base}	Time when T_{base} = 10µs
	Single-bit data transmission		
`t0` (a→b)	Time until end of active low-phase when writing a `'1'` or when reading	1	10µs
`t1` (a→c)	Time until controller samples bus state (read operation)	2	20µs
`t2` (a→d)	Time until end of bit time slot (when writing a `'0'` or when reading)	7	70µs
`t3` (a→e)	Time until end of inter-slot pause (= total duration of one bit)	9	90µs
	Reset pulse and presence detect		
`t4` (f→g)	Time until end of active reset pulse	48	480µs

Symbol	Description	Multiples of T_{base}	Time when T_{base} = 10µs
`t5` (f→h)	Time until controller samples bus presence	55	550µs
`t6` (f→i)	Time until end of presence phase	96	960µs

Default Timing Parameters

The "known-good" default values for base time multiples were chosen for stable and reliable bus operation and not for maximum throughput.

The absolute points in time are hardwired by the VHDL code and cannot be changed during runtime. However, the timing parameter can be customized (if necessary) by editing the ONEWIRE's VHDL source file. The times t0 to t6 correspond to the previous timing diagrams.

Listing 10. Hardwired timing configuration in `neorv32_onewire.vhd`

```
-- timing configuration (absolute time in multiples of the base tick time t_base) --
constant t_write_one_c       : unsigned(6 downto 0) := to_unsigned( 1, 7); -- t0
constant t_read_sample_c     : unsigned(6 downto 0) := to_unsigned( 2, 7); -- t1
constant t_slot_end_c        : unsigned(6 downto 0) := to_unsigned( 7, 7); -- t2
constant t_pause_end_c       : unsigned(6 downto 0) := to_unsigned( 9, 7); -- t3
constant t_reset_end_c       : unsigned(6 downto 0) := to_unsigned(48, 7); -- t4
constant t_presence_sample_c : unsigned(6 downto 0) := to_unsigned(55, 7); -- t5
constant t_presence_end_c    : unsigned(6 downto 0) := to_unsigned(96, 7); -- t6
```

Overdrive Mode

The ONEWIRE controller does not support the overdrive mode natively. However, it can be implemented by reducing the base time $\mathbf{T_{base}}$ (and by eventually changing the hardwired timing configuration in the VHDL source file).

Interrupt

A single interrupt is provided by the ONEWIRE module to signal "idle" condition to the CPU. Whenever the controller is idle (again) and the data/command FIFO is empty, the interrupt becomes active.

Register Map

Table 27. ONEWIRE register map (`struct NEORV32_ONEWIRE`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff20000`	`CTRL`	`0` `ONEWIRE_CTRL_EN`	r/w	ONEWIRE enable, reset if cleared
		`1` `ONEWIRE_CTRL_CLEAR`	-/w	clear RXT FIFO, auto-clears
		`3:2` `ONEWIRE_CTRL_PRSC1 : ONEWIRE_CTRL_PRSC0`	r/w	2-bit clock prescaler select
		`11:4` `ONEWIRE_CTRL_CLKDIV7 : ONEWIRE_CTRL_CLKDIV0`	r/w	8-bit clock divider value
		`14:12` -	r/-	*reserved*, read as zero
		`18:15` `ONEWIRE_CTRL_FIFO_MSB : ONEWIRE_CTRL_FIFO_LSB`	r/-	FIFO depth; log2(`IO_ONEWIRE_FIFO`)
		`27:19` -	r/-	*reserved*, read as zero
		`28` `ONEWIRE_CTRL_TX_FULL`	r/-	TX FIFO full
		`29` `ONEWIRE_CTRL_RX_AVAIL`	r/-	RX FIFO data available
		`30` `ONEWIRE_CTRL_SENSE`	r/-	current state of the bus line
		`31` `ONEWIRE_CTRL_BUSY`	r/-	operation in progress when set or TX FIFO not empty
`0xfff20004`	`DCMD`	`7:0` `ONEWIRE_DCMD_DATA_MSB : ONEWIRE_DCMD_DATA_LSB`	r/w	receive/transmit data
		`9:8` `ONEWIRE_DCMD_CMD_HI : ONEWIRE_DCMD_CMD_LO`	-/w	operation command LSBs
		`10` `ONEWIRE_DCMD_PRESENCE`	-/w	bus presence detected
		`31:11` -	r/-	*reserved*, read as zero

2.8.20. Pulse-Width Modulation Controller (PWM)

Hardware source files:	neorv32_pwm.vhd	
Software driver files:	neorv32_pwm.c	Online software reference (Doxygen)
	neorv32_pwm.h	Online software reference (Doxygen)
Top entity ports:	`pwm_o`	PWM output channels (16-bit)
Configuration generics:	`IO_PWM_NUM_CH`	number of PWM channels to implement (0..16)
CPU interrupts:	none	

Overview

The PWM module implements a pulse-width modulation controller with up to 16 independent channels. Duty cycle and carrier frequency can be programmed individually for each channel.The total number of implemented channels is defined by the `IO_PWM_NUM_CH` generic. The PWM output signal `pwm_o` has a static size of 16-bit. Channel 0 corresponds to bit 0, channel 1 to bit 1 and so on. If less than 16 channels are configured, only the LSB-aligned channel bits are connected while the remaining ones are hardwired to zero.

Theory of Operation

Depending on the configured number channels, the PWM module provides 16 configuration registers `CHANNEL_CFG[0]` to `CHANNEL_CFG[15]` - one for each channel. Regardless of the configuration of `IO_PWM_NUM_CH` all channel registers can be accessed without raising an exception. However, registers above `IO_PWM_NUM_CH-1` are read-only and hardwired to all-zero.

Each configuration provides a 1-bit enable flag to enable/disable the according channel, an 8-bit register for setting the duty cycle and a 3-bit clock prescaler select as well as a 10-bit clock diver for *coarse* and *fine* tuning of the carrier frequency, respectively.

A channel is enabled by setting the `PWM_CFG_EN` bit. If this bit is cleared the according PWM output is set to zero. The duty cycle is programmed via the 8 `PWM_CFG_DUTY` bits. Based on the value programmed to this bits the duty cycle the resulting duty cycle of the according channel can be computed by the following formula:

Duty Cycle[%] = `PWM_CFG_DUTY` / 2^8

The PWM period (carrier frequency) is derived from the processor's main clock (f_{main}). The `PWM_CFG_PRSC` register bits allow to select one out of eight pre-defined clock prescalers for a coarse clock scaling. The 10 `PWM_CFG_CDIV` register bits can be used to apply another fine clock scaling.

Table 28. PWM prescaler configuration

`PWM_CFG_PRSC`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

The resulting PWM carrier frequency is defined by:

f_{PWM}[Hz] = f_{main}[Hz] / (2^8 * `clock_prescaler` * (1 + `PWM_CFG_CDIV`))

Register Map

Table 29. PWM register map (`struct NEORV32_PWM`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff00000`	`CHANNEL_CFG[0]`	`31` - `PWM_CFG_EN`	r/w	Channel 0: channel enabled when set
		`30:28` - `PWM_CFG_PRSC_MSB:PWM_CFG_PRSC_LSB`	r/w	Channel 0: 3-bit clock prescaler select
		`27:18`	r/-	Channel 0: *reserved*, hardwired to zero
		`17:8` - `PWM_CFG_CDIV_MSB:PWM_CFG_CDIV_LSB`	r/w	Channel 0: 10-bit clock divider
		`7:0` - `PWM_CFG_DUTY_MSB:PWM_CFG_DUTY_LSB`	r/w	Channel 0: 8-bit duty cycle
`0xfff00004` … `0xfff00038`	`CHANNEL_CFG[1]` … `CHANNEL_CFG[14]`	…	r/w	Channels 1 to 14
`0xfff0003C`	`CHANNEL_CFG[15]`	`31` - `PWM_CFG_EN`	r/w	Channel 15: channel enabled when set
		`30:28` - `PWM_CFG_PRSC_MSB:PWM_CFG_PRSC_LSB`	r/w	Channel 15: 3-bit clock prescaler select
		`27:18`	r/-	Channel 15: *reserved*, hardwired to zero
		`17:8` - `PWM_CFG_CDIV_MSB:PWM_CFG_CDIV_LSB`	r/w	Channel 15: 10-bit clock divider
		`7:0` - `PWM_CFG_DUTY_MSB:PWM_CFG_DUTY_LSB`	r/w	Channel 15: 8-bit duty cycle

2.8.21. True Random-Number Generator (TRNG)

Hardware source files:	neorv32_trng.vhd	
Software driver files:	neorv32_trng.c	Online software reference (Doxygen)
	neorv32_trng.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	`IO_TRNG_EN`	implement TRNG when `true`
	`IO_TRNG_FIFO`	data FIFO depth, min 1, has to be a power of two

Overview

The NEORV32 true random number generator provides *physically* true random numbers. It is based on free-running ring-oscillators that generate **phase noise** when being sampled by a constant clock. This phase noise is used as physical entropy source. The TRNG features a platform independent architecture without FPGA-specific primitives, macros or attributes so it can be synthesized for *any* FPGA.

In-Depth Documentation

For more information about the neoTRNG architecture and an analysis of its random quality check out the neoTRNG repository: https://github.com/stnolting/neoTRNG

Inferring Latches

The synthesis tool might emit warnings regarding **inferred latches** or **combinatorial loops**. However, this is not design flaw as this is exactly what we want. ;)

Theory of Operation

The TRNG provides two memory mapped interface register. One control register (`CTRL`) for configuration and status check and one data register (`DATA`) for obtaining the random data. The TRNG is enabled by setting the control register's `TRNG_CTRL_EN`. As soon as the `TRNG_CTRL_AVAIL` bit is set a new random data byte is available and can be obtained from the lowest 8 bits of the `DATA` register. If this bit is cleared, there is no valid data available and the reading `DATA` will return all-zero.

An internal entropy FIFO can be configured using the `IO_TRNG_FIFO` generic. This FIFO automatically samples new random data from the TRNG to provide some kind of *random data pool* for applications which require a larger number of random data in a short time. The random data FIFO can be cleared at any time either by disabling the TRNG or by setting the `TRNG_CTRL_FIFO_CLR` flag. The FIFO depth can be retrieved by software via the `TRNG_CTRL_FIFO_*` bits.

Simulation

When simulating the processor the TRNG is automatically set to "simulation mode". In this mode the physical entropy sources (the ring oscillators) are replaced by a simple **pseudo RNG** based on a LFSR providing only **deterministic pseudo-random** data. The `TRNG_CTRL_SIM_MODE` flag of the control register is set if simulation mode is active.

Register Map

Table 30. TRNG register map (`struct NEORV32_TRNG`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfffa0000`	`CTRL`	`0` `TRNG_CTRL_EN`	r/w	TRNG enable
		`1` `TRNG_CTRL_FIFO_CLR`	-/w	flush random data FIFO when set; flag auto-clears
		`5:2` `TRNG_CTRL_FIFO_MSB : TRNG_CTRL_FIFO_LSB`	r/-	FIFO depth, log2(`IO_TRNG_FIFO`)
		`6` `TRNG_CTRL_SIM_MODE`	r/-	simulation mode (PRNG!)
		`7` `TRNG_CTRL_AVAIL`	r/-	random data available when set
`0xfffa0004`	`DATA`	`7:0` `TRNG_DATA_MSB : TRNG_DATA_LSB`	r/-	random data byte
		`31:8` *reserved*	r/-	reserved, read as zero

2.8.22. Custom Functions Subsystem (CFS)

Hardware source files:	neorv32_cfs.vhd	
Software driver files:	neorv32_cfs.c	Online software reference (Doxygen)
	neorv32_cfs.h	Online software reference (Doxygen)
Top entity ports:	`cfs_in_i`	custom input conduit
	`cfs_out_o`	custom output conduit
Configuration generics:	`IO_CFS_EN`	implement CFS when `true`
	`IO_CFS_CONFIG`	custom generic conduit
	`IO_CFS_IN_SIZE`	size of `cfs_in_i`
	`IO_CFS_OUT_SIZE`	size of `cfs_out_o`
CPU interrupts:	fast IRQ channel 1	CFS interrupt (see Processor Interrupts)

Overview

The custom functions subsystem is meant for implementing custom tightly-coupled co-processors or interfaces. IT provides up to 16384 32-bit memory-mapped read/write registers (`REG`, see register map below) that can be accessed by the CPU via normal load/store operations. The actual functionality of these register has to be defined by the hardware designer. Furthermore, the CFS provides two IO conduits to implement custom on-chip or off-chip interfaces.

Just like any other externally-connected IP, logic implemented within the custom functions subsystem can operate *independently* of the CPU providing true parallel processing capabilities. Potential use cases might include dedicated hardware accelerators for en-/decryption (AES), signal processing (FFT) or AI applications (CNNs) as well as custom IO systems like fast memory interfaces (DDR) and mass storage (SDIO), networking (CAN) or real-time data transport (I2S).

If you like to implement *custom instructions* that are executed right within the CPU's ALU see the `Zxcfu` ISA Extension and the according Custom Functions Unit (CFU).

Take a look at the template CFS VHDL source file (`rtl/core/neorv32_cfs.vhd`). The file is highly commented to illustrate all aspects that are relevant for implementing custom CFS-based co-processor designs.

The CFS can also be used to *replicate* existing NEORV32 modules - for example to implement several TWI controllers.

CFS Software Access

The CFS memory-mapped registers can be accessed by software using the provided C-language aliases (see register map table below). Note that all interface registers are defined as 32-bit words of type `uint32_t`.

Listing 11. CFS Software Access Example

```
// C-code CFS usage example
NEORV32_CFS->REG[0] = (uint32_t)some_data_array(i); // write to CFS register 0
int temp = (int)NEORV32_CFS->REG[20]; // read from CFS register 20
```

CFS Interrupt

The CFS provides a single high-level-triggered interrupt request signal mapped to the CPU's fast interrupt channel 1.

CFS Configuration Generic

By default, the CFS provides a single 32-bit `std_ulogic_vector` configuration generic `IO_CFS_CONFIG` that is available in the processor's top entity. This generic can be used to pass custom configuration options from the top entity directly down to the CFS. The actual definition of the generic and it's usage inside the CFS is left to the hardware designer.

CFS Custom IOs

By default, the CFS also provides two unidirectional input and output conduits `cfs_in_i` and `cfs_out_o`. These signals are directly propagated to the processor's top entity. These conduits can be used to implement application-specific interfaces like memory or peripheral connections. The actual use case of these signals has to be defined by the hardware designer.

The size of the input signal conduit `cfs_in_i` is defined via the top's `IO_CFS_IN_SIZE` configuration generic (default = 32-bit). The size of the output signal conduit `cfs_out_o` is defined via the top's `IO_CFS_OUT_SIZE` configuration generic (default = 32-bit). If the custom function subsystem is not implemented (`IO_CFS_EN` = false) the `cfs_out_o` signal is tied to all-zero.

If the CFU output signals are to be used outside the chip, it is recommended to register these signals.

Register Map

Table 31. CFS register map (`struct NEORV32_CFS`)

Address	Name [C]	Bit(s)	R/W	Function
`0xffeb0000`	`REG[0]`	`31:0`	(r)/(w)	custom CFS register 0
`0xffeb0004`	`REG[1]`	`31:0`	(r)/(w)	custom CFS register 1
…	…	`31:0`	(r)/(w)	…
`0xffebfff8`	`REG[16382]`	`31:0`	(r)/(w)	custom CFS register 16382
`0xffebfffc`	`REG[16383]`	`31:0`	(r)/(w)	custom CFS register 16383

2.8.23. Smart LED Interface (NEOLED)

Hardware source files:	neorv32_neoled.vhd	
Software driver files:	neorv32_neoled.c	Online software reference (Doxygen)
	neorv32_neoled.h	Online software reference (Doxygen)
Top entity ports:	`neoled_o`	1-bit serial data output
Configuration generics:	`IO_NEOLED_EN`	implement NEOLED controller when `true`
	`IO_NEOLED_TX_FIFO`	TX FIFO depth, has to be a power of 2, min 1
CPU interrupts:	fast IRQ channel 9	configurable NEOLED data FIFO interrupt (see Processor Interrupts)

Overview

The NEOLED module provides a dedicated interface for "smart RGB LEDs" like WS2812, WS2811 or any other compatible LEDs. These LEDs provide a single-wire interface that uses an asynchronous serial protocol for transmitting color data. Using the NEOLED module allows CPU-independent operation of an arbitrary number of smart LEDs. A configurable data buffer (FIFO) allows to utilize block transfer operation without requiring the CPU.

The NEOLED interface is compatible to the "Adafruit Industries NeoPixel™" products, which feature WS2812 (or older WS2811) smart LEDs. Other LEDs might be compatible as well when adjusting the controller's programmable timing configuration.

The interface provides a single 1-bit output `neoled_o` to drive an arbitrary number of cascaded LEDs. Since the NEOLED module provides 24-bit and 32-bit operating modes, a mixed setup with RGB LEDs (24-bit color) and RGBW LEDs (32-bit color including a dedicated white LED chip) is possible.

Theory of Operation

The NEOLED modules provides two accessible interface registers: the control register `CTRL` and the write-only TX data register `DATA`. The NEOLED module is globally enabled via the control register's `NEOLED_CTRL_EN` bit. Clearing this bit will terminate any current operation, clear the TX buffer, reset the module and set the `neoled_o` output to zero. The precise timing (e.g. implementing the **WS2812** protocol) and transmission mode are fully programmable via the `CTRL` register to provide maximum flexibility.

RGB / RGBW Configuration

NeoPixel™ LEDs are available in two "color" version: LEDs with three chips providing RGB color and LEDs with four chips providing RGB color plus a dedicated white LED chip (= RGBW). Since the intensity of every LED chip is defined via an 8-bit value the RGB LEDs require a frame of 24-bit per module and the RGBW LEDs require a frame of 32-bit per module.

The data transfer quantity of the NEOLED module can be programmed via the NEOLED_MODE_EN control register bit. If this bit is cleared, the NEOLED interface operates in 24-bit mode and will transmit bits 23:0 of the data written to DATA to the LEDs. If NEOLED_MODE_EN is set, the NEOLED interface operates in 32-bit mode and will transmit bits 31:0 of the data written to DATA to the LEDs.

The mode bit can be reconfigured before writing a new data word to DATA in order to support an arbitrary setup/mixture of RGB and RGBW LEDs.

Protocol

The interface of the WS2812 LEDs uses an 800kHz carrier signal. Data is transmitted in a serial manner starting with LSB-first. The intensity for each R, G & B (& W) LED chip (= color code) is defined via an 8-bit value. The actual data bits are transferred by modifying the duty cycle of the signal (the timings for the WS2812 are shown below). A RESET command is "send" by pulling the data line LOW for at least 50µs.

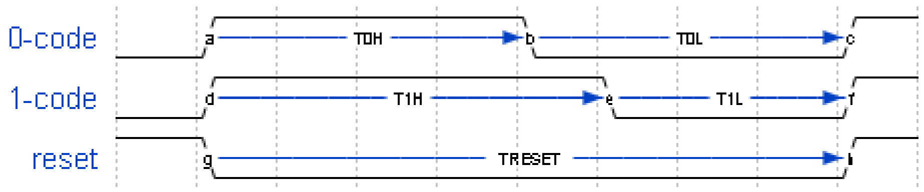


Figure 8. WS2812 bit-level timing (timing does not scale)

Table 32. WS2812 interface timing

T_{total} ($T_{carrier}$)	1.25µs +/- 300ns	period for a single bit
T_{0H}	0.4µs +/- 150ns	high-time for sending a 1
T_{0L}	0.8µs +/- 150ns	low-time for sending a 1
T_{1H}	0.85µs +/- 150ns	high-time for sending a 0
T_{1L}	0.45µs +/- 150 ns	low-time for sending a 0
RESET	Above 50µs	low-time for sending a RESET command

Timing Configuration

The basic carrier frequency (800kHz for the WS2812 LEDs) is configured via a 3-bit main clock prescaler (NEOLED_CTRL_PRSC*, see table below) that scales the main processor clock f_{main} and a 5-bit cycle multiplier NEOLED_CTRL_T_TOT_*.

Table 33. NEOLED Prescaler Configuration

NEOLED_CTRL_PRSCx	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Resulting clock_prescaler	2	4	8	64	128	1024	2048	4096

The duty-cycles (or more precisely: the high- and low-times for sending either a '1' bit or a '0' bit) are defined via the 5-bit NEOLED_CTRL_T_ONE_H_* and NEOLED_CTRL_T_ZERO_H_* values, respectively.

These programmable timing constants allow to adapt the interface for a wide variety of smart LED protocol (for example WS2812 vs. WS2811).

Timing Configuration - Example (WS2812)

Generate the base clock f_{TX} for the NEOLED TX engine:

- processor clock f_{main} = 100 MHz
- `NEOLED_CTRL_PRSCx` = `0b001` = f_{main} / 4

$\boldsymbol{f_{TX}}$ = $f_{main}[Hz]$ / `clock_prescaler` = 100MHz / 4 = 25MHz

$\boldsymbol{T_{TX}}$ = 1 / $\boldsymbol{f_{TX}}$ = 40ns

Generate carrier period ($T_{carrier}$) and **high-times** (duty cycle) for sending `0` (T_{0H}) and `1` (T_{1H}) bits:

- `NEOLED_CTRL_T_TOT` = `0b11110` (= decimal 30)
- `NEOLED_CTRL_T_ZERO_H` = `0b01010` (= decimal 10)
- `NEOLED_CTRL_T_ONE_H` = `0b10100` (= decimal 20)

$\boldsymbol{T_{carrier}}$ = $\boldsymbol{T_{TX}}$ * `NEOLED_CTRL_T_TOT` = 40ns * 30 = 1.4µs

$\boldsymbol{T_{0H}}$ = $\boldsymbol{T_{TX}}$ * `NEOLED_CTRL_T_ZERO_H` = 40ns * 10 = 0.4µs

$\boldsymbol{T_{1H}}$ = $\boldsymbol{T_{TX}}$ * `NEOLED_CTRL_T_ONE_H` = 40ns * 20 = 0.8µs

The NEOLED SW driver library (`neorv32_neoled.h`) provides a simplified configuration function that configures all timing parameters for driving WS2812 LEDs based on the processor clock frequency.

TX Data FIFO

The interface features a configurable TX data buffer (a FIFO) to allow more CPU-independent operation. The buffer depth is configured via the `IO_NEOLED_TX_FIFO` top generic (default = 1 entry). The FIFO size configuration can be read via the `NEOLED_CTRL_BUFS_x` control register bits, which result log2(*IO_NEOLED_TX_FIFO*).

When writing data to the `DATA` register the data is automatically written to the TX buffer. Whenever data is available in the buffer the serial transmission engine will take and transmit it to the LEDs. The data transfer size (`NEOLED_MODE_EN`) can be modified at any time since this control register bit is also buffered in the FIFO. This allows an arbitrary mix of RGB and RGBW LEDs in the chain.

Software can check the FIFO fill level via the control register's `NEOLED_CTRL_TX_EMPTY`, `NEOLED_CTRL_TX_HALF` and `NEOLED_CTRL_TX_FULL` flags. The `NEOLED_CTRL_TX_BUSY` flags provides additional information if the the serial transmit engine is still busy sending data.

Please note that the timing configurations (`NEOLED_CTRL_PRSCx`, `NEOLED_CTRL_T_TOT_x`, `NEOLED_CTRL_T_ONE_H_x` and `NEOLED_CTRL_T_ZERO_H_x`) are **NOT** stored to the buffer.

> Changing these value while the buffer is not empty or the TX engine is still busy will cause data corruption.

Strobe Command ("RESET")

According to the WS2812 specs the data written to the LED's shift registers is strobed to the actual PWM driver registers when the data line is low for 50µs ("RESET" command, see table above). This can be implemented using busy-wait for at least 50µs. Obviously, this concept wastes a lot of processing power.

To circumvent this, the NEOLED module provides an option to automatically issue an idle time for creating the RESET command. If the `NEOLED_CTRL_STROBE` control register bit is set, *all* data written to the data FIFO (via `DATA`, the actually written data is irrelevant) will trigger an idle phase (`neoled_o` = zero) of 127 periods (= $\boldsymbol{T}_{carrier}$). This idle time will cause the LEDs to strobe the color data into the PWM driver registers.

Since the `NEOLED_CTRL_STROBE` flag is also buffered in the TX buffer, the RESET command is treated just as another data word being written to the TX buffer making busy wait concepts obsolete and allowing maximum refresh rates.

NEOLED Interrupt

The NEOLED modules features a single interrupt that triggers based on the current TX buffer fill level. The interrupt can only become pending if the NEOLED module is enabled. The specific interrupt condition is configured via the `NEOLED_CTRL_IRQ_CONF` bit in the unit's control register.

If `NEOLED_CTRL_IRQ_CONF` is set, the module's interrupt is generated whenever the TX FIFO is less than half-full. In this case software can write up to `IO_NEOLED_TX_FIFO`/2 new data words to `DATA` without checking the FIFO status flags. If `NEOLED_CTRL_IRQ_CONF` is cleared, an interrupt is generated when the TX FIFO is empty.

Once the NEOLED interrupt has fired it remains pending until the actual cause of the interrupt is resolved.

Register Map

Table 34. NEOLED register map (`struct NEORV32_NEOLED`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfffd0000`	`CTRL`	`0` `NEOLED_CTRL_EN`	r/w	NEOLED enable
		`1` `NEOLED_CTRL_MODE`	r/w	data transfer size; `0`=24-bit; `1`=32-bit
		`2` `NEOLED_CTRL_STROBE`	r/w	`0`=send normal color data; `1`=send RESET command on data write access
		`5:3` `NEOLED_CTRL_PRSC2 : NEOLED_CTRL_PRSC0`	r/w	3-bit clock prescaler, bit 0
		`9:6` `NEOLED_CTRL_BUFS3 : NEOLED_CTRL_BUFS0`	r/-	4-bit log2(*IO_NEOLED_TX_FIFO*)
		`14:10` `NEOLED_CTRL_T_TOT_4 : NEOLED_CTRL_T_TOT_0`	r/w	5-bit pulse clock ticks per total single-bit period (T_{total})
		`19:15` `NEOLED_CTRL_T_ZERO_H_4 : NEOLED_CTRL_T_ZERO_H_0`	r/w	5-bit pulse clock ticks per high-time for sending a zero-bit (T_{0H})
		`24:20` `NEOLED_CTRL_T_ONE_H_4 : NEOLED_CTRL_T_ONE_H_0`	r/w	5-bit pulse clock ticks per high-time for sending a one-bit (T_{1H})
		`27` `NEOLED_CTRL_IRQ_CONF`	r/w	TX FIFO interrupt configuration: `0`=IRQ if FIFO is empty, `1`=IRQ if FIFO is less than half-full
		`28` `NEOLED_CTRL_TX_EMPTY`	r/-	TX FIFO is empty
		`29` `NEOLED_CTRL_TX_HALF`	r/-	TX FIFO is *at least* half full
		`30` `NEOLED_CTRL_TX_FULL`	r/-	TX FIFO is full
		`31` `NEOLED_CTRL_TX_BUSY`	r/-	TX serial engine is busy when set
`0xfffd0004`	`DATA`	`31:0` / `23:0`	-/w	TX data (32- or 24-bit, depending on *NEOLED_CTRL_MODE* bit)

2.8.24. General Purpose Timer (GPTMR)

Hardware source files:	neorv32_gptmr.vhd	
Software driver files:	neorv32_gptmr.c	Online software reference (Doxygen)
	neorv32_gptmr.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	`IO_GPTMR_EN`	implement general purpose timer when `true`
CPU interrupts:	fast IRQ channel 12	timer interrupt (see Processor Interrupts)

Overview

The general purpose timer module implements a simple yet universal 32-bit timer. It is implemented if the processor's `IO_GPTMR_EN` top generic is set `true`. The timer provides a pre-scaled counter register that can trigger an interrupt when reaching a programmable threshold value.

The GPTMR provides three interface registers : a control register (`CTRL`), a 32-bit counter register (`COUNT`) and a 32-bit threshold register (`THRES`). The timer is globally enabled by setting the `GPTMR_CTRL_EN` bit in the module's control register. When the timer is enable the `COUNT` register will start incrementing from zero at a programmable rate that scales the main processor clock. this pre-scaler is configured via the three `GPTMR_CTRL_PRSCx` control register bits:

Table 35. GPTMR prescaler configuration

`GPTMR_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

Whenever the counter register `COUNT` equals the programmable threshold value `THRES` the module's interrupt signal becomes pending (indicated by `GPTMR_CTRL_IRQ_PND` being set). Note that a pending interrupt has to be cleared manually by writing a `1` to `GPTMR_CTRL_IRQ_CLR`.

The control register's `GPTMR_CTRL_MODE` bit defines what will happen when `COUNT == THRES`.

* `GPTMR_CTRL_MODE = 0`: **single-shot mode** - the `COUNT` register will stop incrementing
* `GPTMR_CTRL_MODE = 1`: **continuous mode** - the `COUNT` register is automatically reset and restarts incrementing from zero

Resetting the Counter

Disabling the GPTMR will also clear the `COUNT` register.

Interrupt

The GPTRM provides a single interrupt line is triggered whenever `COUNT` equals `THRES`. Once triggered, the interrupt will stay pending until explicitly cleared by writing a 1 to

GPTMR_CTRL_IRQ_CLR.

Register Map

Table 36. GPTMR register map (`struct NEORV32_GPTMR`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xfff10000`	`CTRL`	`0` `GPTMR_CTRL_EN`	r/w	Timer enable flag
		`3:1` `GPTMR_CTRL_PRSC2 : GPTMR_CTRL_PRSC0`	r/w	3-bit clock prescaler select
		`4` `GPTMR_CTRL_MODE`	r/w	Operation mode (0=single-shot, 1=continuous)
		`29:5` -	r/-	*reserved*, read as zero
		`30` `GPTMR_CTRL_IRQ_CLR`	-/w	Write `1` to clear timer-match interrupt; auto-clears
		`31` `GPTMR_CTRL_IRQ_PND`	r/-	Timer-match interrupt pending
`0xfff10004`	`THRES`	`31:0`	r/w	Threshold value register
`0xfff10008`	`COUNT`	`31:0`	r/-	Counter register

2.8.25. Execute In Place Module (XIP)

Hardware source files:	neorv32_xip.vhd	XIP module
	neorv32_cache.vhd	Generic cache module
Software driver files:	neorv32_xip.c	Online software reference (Doxygen)
	neorv32_xip.h	Online software reference (Doxygen)
Top entity ports:	`xip_csn_o`	1-bit chip select, low-active
	`xip_clk_o`	1-bit serial clock output
	`xip_dat_i`	1-bit serial data input
	`xip_dat_o`	1-bit serial data output
Configuration generics:	`XIP_EN`	implement XIP module when `true`
	`XIP_CACHE_EN`	implement XIP cache when `true`
	`XIP_CACHE_NUM_BLOCKS`	number of blocks in XIP cache; has to be a power of two
	`XIP_CACHE_BLOCK_SIZE`	number of bytes per XIP cache block; has to be a power of two, min 4
CPU interrupts:	none	

Overview

The execute in-place (XIP) module allows to execute code (and read constant data) directly from an external SPI flash memory. The standard serial peripheral interface (SPI) is used as transfer protocol. All bus requests issued by the CPU are converted transparently into SPI flash access commands. Hence, the external XIP flash behaves like a simple on-chip ROM.

From the CPU side, the modules provides two independent interfaces: one for transparently accessing the XIP flash and another one for accessing the module's control and status registers. The first interface provides the *transparent* gateway to the SPI flash, so the CPU can directly fetch and execute instructions and/or read constant data. Note that this interface is read-only. Any write access will raise a bus error exception. The second interface is mapped to the processor's IO space and allows accesses to the XIP module's configuration registers as well as conducting individual SPI transfers.

The XIP module provides an optional configurable cache to accelerate SPI flash accesses.

XIP Address Mapping

When XIP mode is enabled the flash is mapped to fixed address space region starting at address `0xE0000000` (see section Address Space) supporting a maximum flash size of 256MB.

XIP Example Program

An example program is provided in `sw/example/demo_xip` that illustrate how to program and configure an external SPI flash to run a program from it.

SPI Configuration

The XIP module accesses external flash using the standard SPI protocol. The module always sends data MSB-first and provides all of the standard four clock modes (0..3), which are configured via the `XIP_CTRL_CPOL` (clock polarity) and `XIP_CTRL_CPHA` (clock phase) control register bits, respectively. The flash's "read command", which initiates a read access, is defined by the `XIP_CTRL_RD_CMD` control register bits. For most SPI flash memories this is `0x03` for *normal* SPI mode.

The SPI clock (`xip_clk_o`) frequency is programmed by the 3-bit `XIP_CTRL_PRSCx` clock prescaler for a coarse clock selection and a 4-bit clock divider `XPI_CTRL_CDIVx` for a fine clock selection. The following clock prescalers (`XIP_CTRL_PRSCx`) are available:

Table 37. XIP clock prescaler configuration

`XIP_CTRL_PRSCx`	`0b000`	`0b001`	`0b010`	`0b011`	`0b100`	`0b101`	`0b110`	`0b111`
Resulting `clock_prescaler`	2	4	8	64	128	1024	2048	4096

Based on the programmed clock configuration, the actual SPI clock frequency f_{SPI} is derived from the processor's main clock f_{main} according to the following equation:

$f_{SPI} = f_{main}[Hz]$ / (2 * `clock_prescaler` * (1 + `XPI_CTRL_CDIVx`))

Hence, the maximum SPI clock is f_{main} / 4 and the lowest SPI clock is f_{main} / 131072. The SPI clock is always symmetric having a duty cycle of 50%.

High-Speed Mode

The XIP module provides a high-speed mode to further boost the maximum SPI clock frequency. When enabled via the control register's `XIP_CTRL_HIGHSPEED` bit the clock prescaler configuration (`XIP_CTRL_PRSCx` bits) is overridden setting it to a minimal factor of 1. However, the clock speed can still be fine-tuned using the `XPI_CTRL_CDIVx` bits.

$f_{SPI} = f_{main}[Hz]$ / (2 * 1 * (1 + `XPI_CTRL_CDIVx`))

Hence, the maximum SPI clock when in high-speed mode is f_{main} / 2.

Direct SPI Access

The XIP module allows to initiate *direct* SPI transactions. This feature can be used to configure the attached SPI flash or to perform direct read and write accesses to the flash memory. Two data registers `DATA_LO` and `DATA_HI` are provided to send up to 64-bit of SPI data. The `DATA_HI` register is write-only, so a total of just 32-bits of receive data is provided. Note that the module handles the chip-select line (`xip_csn_o`) by itself so it is not possible to construct larger consecutive transfers.

The actual data transmission size in bytes is defined by the control register's `XIP_CTRL_SPI_NBYTES` bits. Any configuration from 1 byte to 8 bytes is valid. Other value will result in unpredictable behavior.

Since data is always transferred MSB-first, the data in `DATA_HI:DATA_LO` also has to be MSB-aligned. Receive data is available in `DATA_LO` only since `DATA_HI` is write-only. Writing to `DATA_HI` triggers the actual SPI transmission. The `XIP_CTRL_PHY_BUSY` control register flag indicates a transmission being in progress.

The chip-select line of the XIP module (`xip_csn_o`) will only become asserted (enabled, pulled low) if the `XIP_CTRL_SPI_CSEN` control register bit is set. If this bit is cleared, `xip_csn_o` is always disabled (pulled high).

Direct SPI mode is only possible when the module is enabled (setting `XIP_CTRL_EN`) but **before** the actual XIP mode is enabled via `XIP_CTRL_XIP_EN`.

When the XIP mode is not enabled, the XIP module can also be used as additional general purpose SPI controller with a transfer size of up to 64 bits per transmission.

Using the XIP Mode

The XIP module is globally enabled by setting the `XIP_CTRL_EN` bit in the device's `CTRL` control register. Clearing this bit will reset the whole module and will also terminate any pending SPI transfer.

Since there is a wide variety of SPI flash components with different sizes, the XIP module allows to specify the address width of the flash: the number of address bytes used for addressing flash memory content has to be configured using the control register's *XIP_CTRL_XIP_ABYTES* bits. These two bits contain the number of SPI address bytes (**minus one**). For example for a SPI flash with 24-bit addresses these bits have to be set to `0b10`.

The transparent XIP accesses are transformed into SPI transmissions with the following format (starting with the MSB):

- 8-bit command: configured by the `XIP_CTRL_RD_CMD` control register bits ("SPI read command")
- 8 to 32 bits address: defined by the `XIP_CTRL_XIP_ABYTES` control register bits ("number of address bytes")
- 32-bit data: sending zeros and receiving the according flash word (32-bit)

Hence, the maximum XIP transmission size is 72-bit, which has to be configured via the `XIP_CTRL_SPI_NBYTES` control register bits. Note that the 72-bit transmission size is only available in XIP mode. The transmission size of the direct SPI accesses is limited to 64-bit.

When using four SPI flash address bytes, the most significant 4 bits of the address are always hardwired to zero allowing a maximum **accessible** flash size of 256MB.

The XIP module always fetches a full naturally aligned 32-bit word from the SPI

Copyright by Stephan Nolting. All rights reserved.

flash. Any sub-word data masking or alignment will be performed by the CPU core logic.

The XIP mode requires the 4-byte data words in the flash to be ordered in **little-endian** byte order.

After the SPI properties (including the amount of address bytes **and** the total amount of SPI transfer bytes) and XIP address mapping are configured, the actual XIP mode can be enabled by setting the control register's `XIP_CTRL_XIP_EN` bit. This will enable the "transparent SPI access port" of the module and thus, the *transparent* conversion of access requests into proper SPI flash transmissions. Hence, any access to the processor's memory-mapped XIP region (`0xE0000000` to `0xEFFFFFFF`) will be converted into SPI flash accesses. Make sure `XIP_CTRL_SPI_CSEN` is also set so the module can actually select/enable the attached SPI flash. No more direct SPI accesses via `DATA_HI:DATA_LO` are possible when the XIP mode is enabled. However, the XIP mode can be disabled at any time.

If the XIP module is disabled (*XIP_CTRL_EN* = `0`) any accesses to the memory-mapped XIP flash address region will raise a bus access exception. If the XIP module is enabled (*XIP_CTRL_EN* = `1`) but XIP mode is not enabled yet (*XIP_CTRL_XIP_EN* = '0') any access to the programmed XIP memory segment will also raise a bus access exception.

It is highly recommended to enable the Processor-Internal Instruction Cache (iCACHE) to cover some of the SPI access latency.

XIP Cache

Since every single instruction fetch request from the CPU is translated into serial SPI transmissions the access latency is very high resulting in a low throughput. In order to improve performance, the XIP module provides an optional cache that allows to buffer recently-accessed data. The cache is implemented as a simple direct-mapped read-only cache with a configurable cache layout:

- `XIP_CACHE_EN`: when set to `true` the CIP cache is implemented
- `XIP_CACHE_NUM_BLOCKS` defines the number of cache blocks (or lines)
- `XIP_CACHE_BLOCK_SIZE` defines the size in bytes of each cache block

When the cache is implemented, the XIP module operates in **burst mode** utilizing the flash's *incremental read* capabilities. Thus, several bytes (= `XIP_CACHE_BLOCK_SIZE`) are read consecutively from the flash using a single read command.

The XIP cache is cleared when the XIP module is disabled (`XIP_CTRL_EN = 0`), when XIP mode is disabled (`XIP_CTRL_XIP_EN = 0`) or when the CPU issues a `fence[.i]` instruction.

Register Map

Table 38. XIP Register Map (`struct NEORV32_XIP`)

Address	Name [C]	Bit(s), Name [C]	R/W	Function
`0xffff4000`	`CTRL`	`0` `XIP_CTRL_EN`	r/w	XIP module enable
		`3:1` `XIP_CTRL_PRSC2 : XIP_CTRL_PRSC0`	r/w	3-bit SPI clock prescaler select
		`4` `XIP_CTRL_CPOL`	r/w	SPI clock polarity
		`5` `XIP_CTRL_CPHA`	r/w	SPI clock phase
		`9:6` `XIP_CTRL_SPI_NBYTES_MSB : XIP_CTRL_SPI_NBYTES_LSB`	r/w	Number of bytes in SPI transaction (1..9)
		`10` `XIP_CTRL_XIP_EN`	r/w	XIP mode enable
		`12:11` `XIP_CTRL_XIP_ABYTES_MSB : XIP_CTRL_XIP_ABYTES_LSB`	r/w	Number of address bytes for XIP flash (minus 1)
		`20:13` `XIP_CTRL_RD_CMD_MSB : XIP_CTRL_RD_CMD_LSB`	r/w	Flash read command
		`21` `XIP_CTRL_SPI_CSEN`	r/w	Allow SPI chip-select to be actually asserted when set
		`22` `XIP_CTRL_HIGHSPEED`	r/w	enable SPI high-speed mode (ignoring `XIP_CTRL_PRSCx`)
		`26:23` `XIP_CTRL_CDIV3 : XIP_CTRL_CDIV0`	r/-	4-bit clock divider for fine-tuning
		`29:27` -	r/-	*reserved*, read as zero
		`30` `XIP_CTRL_PHY_BUSY`	r/-	SPI PHY busy when set
		`31` `XIP_CTRL_XIP_BUSY`	r/-	XIP access in progress when set
`0xffff4004`	*reserved*	`31:0`	r/-	*reserved*, read as zero
`0xffff4008`	`DATA_LO`	`31:0`	r/w	Direct SPI access - data register low
`0xffff400C`	`DATA_HI`	`31:0`	-/w	Direct SPI access - data register high; write access triggers SPI transfer

2.8.26. System Configuration Information Memory (SYSINFO)

Hardware source files:	neorv32_sysinfo.vhd	
Software driver files:	neorv32_sysinfo.h	Online software reference (Doxygen)
Top entity ports:	none	
Configuration generics:	*	most of the top's configuration generics
CPU interrupts:	none	

Overview

The SYSINFO module allows the application software to determine the setting of most of the Processor Top Entity - Generics that are related to CPU and processor/SoC configuration. This device is always implemented - regardless of the actual hardware configuration since the NEORV32 software framework requires information from this device for correct operation. However, advanced users that do not want to use the default NEORV32 software framework can choose to disable the entire SYSINFO module. This might also be suitable for setups that use the processor just as wrapper for a CPU-only configuration.

Disabling the SYSINFO Module

Setting the `IO_DISABLE_SYSINFO` top entity generic to `true` will remove the SYSINFO module from the design. This option is suitable for advanced uses that wish to use a CPU-only setup that still contains the bus infrastructure. As a result, large parts of the NEORV32 software framework no longer work (e.g. most IO drivers, the RTE and the bootloader). **Hence, this option is not recommended.**

Register Map

All registers of this module are read-only except for the `CLK` register. Upon reset, the `CLK` registers is initialized from the `CLOCK_FREQUENCY` top entity generic. Application software can override this default value in order, for example, to take into account a dynamic frequency scaling of the processor.

Table 39. SYSINFO register map (`struct NEORV32_SYSINFO`)

Address	Name [C]	R/W	Description
`0xfffe0000`	`CLK`	r/w	clock frequency in Hz (initialized from top's `CLOCK_FREQUENCY` generic)
`0xfffe0004`	`MISC[4]`	r/-	miscellaneous system configurations (see SYSINFO - Miscellaneous Configuration)
`0xfffe0008`	`SOC`	r/-	specific SoC configuration (see SYSINFO - SoC Configuration)
`0xfffe000c`	`CACHE`	r/-	cache configuration information (see SYSINFO - Cache Configuration)

SYSINFO - Miscellaneous Configuration

Bit fields in this register are set to all-zero if the according memory system is not implemented.

Table 40. SYSINFO `MEM` Bytes

Byte	Name [C]	Description
`0`	`SYSINFO_MISC_IMEM`	*log2*(internal IMEM size in bytes), via top's `MEM_INT_IMEM_SIZE` generic
`1`	`SYSINFO_MISC_DMEM`	*log2*(internal DMEM size in bytes), via top's `MEM_INT_DMEM_SIZE` generic
`2`	`SYSINFO_MISC_HART`	number of physical CPU cores ("harts")
`3`	`SYSINFO_MISC_BOOT`	boot mode configuration, via top's `BOOT_MODE_SELECT` generic (see Boot Configuration)

SYSINFO - SoC Configuration

Table 41. SYSINFO `SOC` Bits

Bit	Name [C]	Description
`0`	`SYSINFO_SOC_BOOTLOADER`	set if processor-internal bootloader is implemented (via top's `BOOT_MODE_SELECT` generic; see Boot Configuration)
`1`	`SYSINFO_SOC_XBUS`	set if external Wishbone bus interface is implemented (via top's `XBUS_EN` generic)
`2`	`SYSINFO_SOC_MEM_INT_IMEM`	set if processor-internal DMEM is implemented (via top's `MEM_INT_IMEM_EN` generic)
`3`	`SYSINFO_SOC_MEM_INT_DMEM`	set if processor-internal IMEM is implemented (via top's `MEM_INT_DMEM_EN` generic)
`4`	`SYSINFO_SOC_OCD`	set if on-chip debugger is implemented (via top's `OCD_EN` generic)
`5`	`SYSINFO_SOC_ICACHE`	set if processor-internal instruction cache is implemented (via top's `ICACHE_EN` generic)
`6`	`SYSINFO_SOC_DCACHE`	set if processor-internal data cache is implemented (via top's `DCACHE_EN` generic)
`7`	-	*reserved*, read as zero
`8`	`SYSINFO_SOC_XBUS_CACHE`	set if external bus interface cache is implemented (via top's `XBUS_CACHE_EN` generic)
`9`	`SYSINFO_SOC_XIP`	set if XIP module is implemented (via top's `XIP_EN` generic)

Bit	Name [C]	Description
10	SYSINFO_SOC_XIP_CACHE	set if XIP cache is implemented (via top's XIP_CACHE_EN generic)
11	SYSINFO_SOC_OCD_AUTH	set if on-chip debugger authentication is implemented (via top's OCD_AUTHENTICATION generic)
12	SYSINFO_SOC_IMEM_ROM	set if processor-internal IMEM is implemented as pre-initialized ROM (via top's BOOT_MODE_SELECT generic; see Boot Configuration)
13	SYSINFO_SOC_IO_TWD	set if TWD is implemented (via top's IO_TWD_EN generic)
14	SYSINFO_SOC_IO_DMA	set if direct memory access controller is implemented (via top's IO_DMA_EN generic)
15	SYSINFO_SOC_IO_GPIO	set if GPIO is implemented (via top's IO_GPIO_EN generic)
16	SYSINFO_SOC_IO_CLINT	set if CLINT is implemented (via top's IO_CLINT_EN generic)
17	SYSINFO_SOC_IO_UART0	set if primary UART0 is implemented (via top's IO_UART0_EN generic)
18	SYSINFO_SOC_IO_SPI	set if SPI is implemented (via top's IO_SPI_EN generic)
19	SYSINFO_SOC_IO_TWI	set if TWI is implemented (via top's IO_TWI_EN generic)
20	SYSINFO_SOC_IO_PWM	set if PWM is implemented (via top's IO_PWM_NUM_CH generic)
21	SYSINFO_SOC_IO_WDT	set if WDT is implemented (via top's IO_WDT_EN generic)
22	SYSINFO_SOC_IO_CFS	set if custom functions subsystem is implemented (via top's IO_CFS_EN generic)
23	SYSINFO_SOC_IO_TRNG	set if TRNG is implemented (via top's IO_TRNG_EN generic)
24	SYSINFO_SOC_IO_SDI	set if SDI is implemented (via top's IO_SDI_EN generic)
25	SYSINFO_SOC_IO_UART1	set if secondary UART1 is implemented (via top's IO_UART1_EN generic)
26	SYSINFO_SOC_IO_NEOLED	set if NEOLED is implemented (via top's IO_NEOLED_EN generic)
27	-	*reserved*, read as zero
28	SYSINFO_SOC_IO_GPTMR	set if GPTMR is implemented (via top's IO_GPTMR_EN generic)
29	SYSINFO_SOC_IO_SLINK	set if stream link interface is implemented (via top's IO_SLINK_EN generic)

Bit	Name [C]	Description
30	SYSINFO_SOC_IO_ONEWIRE	set if ONEWIRE interface is implemented (via top's IO_ONEWIRE_EN generic)
31	SYSINFO_SOC_IO_CRC	set if cyclic redundancy check unit is implemented (via top's IO_CRC_EN generic)

SYSINFO - Cache Configuration

The SYSINFO cache register provides information about the configuration of the processor caches:

- Processor-Internal Instruction Cache (iCACHE)
- Processor-Internal Data Cache (dCACHE)
- Execute In Place Module (XIP) cache (XIP-CACHE)
- Processor-External Bus Interface (XBUS) cache (XBUS-CACHE)

Table 42. SYSINFO CACHE *Bits*

Bit	Name [C]	Description
3:0	SYSINFO_CACHE_INST_BLOCK_SIZE_3 : SYSINFO_CACHE_INST_BLOCK_SIZE_0	*log2*(i-cache block size in bytes), via top's ICACHE_BLOCK_SIZE generic
7:4	SYSINFO_CACHE_INST_NUM_BLOCKS_3 : SYSINFO_CACHE_INST_NUM_BLOCKS_0	*log2*(i-cache number of cache blocks), via top's ICACHE_NUM_BLOCKS generic
11:8	SYSINFO_CACHE_DATA_BLOCK_SIZE_3 : SYSINFO_CACHE_DATA_BLOCK_SIZE_0	*log2*(d-cache block size in bytes), via top's DCACHE_BLOCK_SIZE generic
15:12	SYSINFO_CACHE_DATA_NUM_BLOCKS_3 : SYSINFO_CACHE_DATA_NUM_BLOCKS_0	*log2*(d-cache number of cache blocks), via top's DCACHE_NUM_BLOCKS generic
19:16	SYSINFO_CACHE_XIP_BLOCK_SIZE_3 : SYSINFO_CACHE_XIP_BLOCK_SIZE_0	*log2*(xip-cache block size in bytes), via top's XIP_CACHE_BLOCK_SIZE generic
23:20	SYSINFO_CACHE_XIP_NUM_BLOCKS_3 : SYSINFO_CACHE_XIP_NUM_BLOCKS_0	*log2*(xip-cache number of cache blocks), via top's XIP_CACHE_NUM_BLOCKS generic
27:24	SYSINFO_CACHE_XBUS_BLOCK_SIZE_3 : SYSINFO_CACHE_XBUS_BLOCK_SIZE_0	*log2*(xbus-cache block size in bytes), via top's XBUS_CACHE_BLOCK_SIZE generic
31:28	SYSINFO_CACHE_XBUS_NUM_BLOCKS_3 : SYSINFO_CACHE_XBUS_NUM_BLOCKS_0	*log2*(xbus-cache number of cache blocks), via top's XBUS_CACHE_NUM_BLOCKS generic

Chapter 3. NEORV32 Central Processing Unit (CPU)

The NEORV32 CPU is an area-optimized RISC-V core implementing the `rv32i_zicsr_zifencei` base (privileged) ISA and supporting several additional/optional ISA extensions. The CPU's micro architecture is based on a von-Neumann machine build upon a mixture of multi-cycle and pipelined execution schemes. Optionally, the core can be implemented as SMP Dual-Core Configuration.

RISC-V Specifications

This chapter assumes that the reader is familiar with the official RISC-V *User* and *Privileged Architecture* specifications.

Section Structure

- RISC-V Compatibility
- CPU Top Entity - Signals and CPU Top Entity - Generics
- Architecture and Full Virtualization
- Instruction Sets and Extensions and Custom Functions Unit (CFU)
- Control and Status Registers (CSRs)
- Traps, Exceptions and Interrupts
- Bus Interface

3.1. RISC-V Compatibility

The NEORV32 CPU passes the tests of the **official RISCOF RISC-V Architecture Test Framework**. This framework is used to check RISC-V implementations for compatibility to the official RISC-V user/privileged ISA specifications. The NEORV32 port of this test framework is available in a separate repository at GitHub: https://github.com/stnolting/neorv32-riscof

Unsupported ISA Extensions

Executing instructions or accessing CSRs from yet unsupported ISA extensions will raise an illegal instruction exception (see section Full Virtualization).

Incompatibility Issues and Limitations

`time[h]` *CSRs (Wall Clock Time)*

The NEORV32 does not implement the `time[h]` registers. Any access to these registers will trap. It is recommended that the trap handler software provides a means of accessing the the machine timer of the Core Local Interruptor (CLINT).

No Hardware Support of Misaligned Memory Accesses

The CPU does not support resolving unaligned memory access by the hardware (this is not a RISC-V-incompatibility issue but an important thing to know!). Any kind of unaligned memory access will raise an exception to allow a *software-based* emulation provided by the application. However, unaligned memory access can be **emulated** using the NEORV32 runtime environment. See section Application Context Handling for more information.

3.2. CPU Top Entity - Signals

The following table shows all interface signals of the CPU top entity `rtl/core/neorv32_cpu.vhd`. The type of all signals is *std_ulogic* or *std_ulogic_vector*, respectively. The "Dir." column shows the signal direction as seen from the CPU.

Table 43. NEORV32 CPU Signal List

Signal	Width/Type	Dir	Description
Global Signals			
`clk_i`	1	in	Global clock line, all registers triggering on rising edge.
`rstn_i`	1	in	Global reset, low-active.
Interrupts (Traps, Exceptions and Interrupts)			
`msi_i`	1	in	RISC-V machine software interrupt.
`mei_i`	1	in	RISC-V machine external interrupt.
`mti_i`	1	in	RISC-V machine timer interrupt.
`firq_i`	16	in	Custom fast interrupt request signals.
`dbi_i`	1	in	Request CPU to halt and enter debug mode (RISC-V On-Chip Debugger (OCD)).
Instruction Bus Interface			
`ibus_req_o`	`bus_req_t`	out	Instruction fetch bus request.
`ibus_rsp_i`	`bus_rsp_t`	in	Instruction fetch bus response.
Data Bus Interface			
`dbus_req_o`	`bus_req_t`	out	Data access (load/store) bus request.
`dbus_rsp_i`	`bus_rsp_t`	in	Data access (load/store) bus response.
Inter-Core Communication (ICC) TX links			
`icc_tx_rdy_o`	2	out	Data available for cores `0..1`.
`icc_tx_ack_i`	2	in	Read-enable from cores `0..1`.
`icc_tx_dat_o`	2*32	out	Data for cores `0..1`.
Inter-Core Communication (ICC) RX links			
`icc_rx_rdy_i`	2	in	Data available from cores `0..1`.
`icc_rx_ack_o`	2	out	Read-enable for cores `0..1`.
`icc_rx_dat_i`	2*32	in	Data from cores `0..1`.

Bus Interface Protocol

See section Bus Interface for the instruction fetch and data access interface protocol and the according interface types (`bus_req_t` and `bus_rsp_t`).

3.3. CPU Top Entity - Generics

Most of the CPU configuration generics are a subset of the actual Processor configuration generics (see section Processor Top Entity - Generics). and are not listed here. However, the CPU provides some *specific* generics that are used to configure the CPU for the NEORV32 processor setup. These generics are assigned by the processor setup only and are not available for user defined configuration. The specific generics are listed below.

Table Abbreviations

The generic type "suv(x:y)" represents a `std_ulogic_vector(x downto y)`.

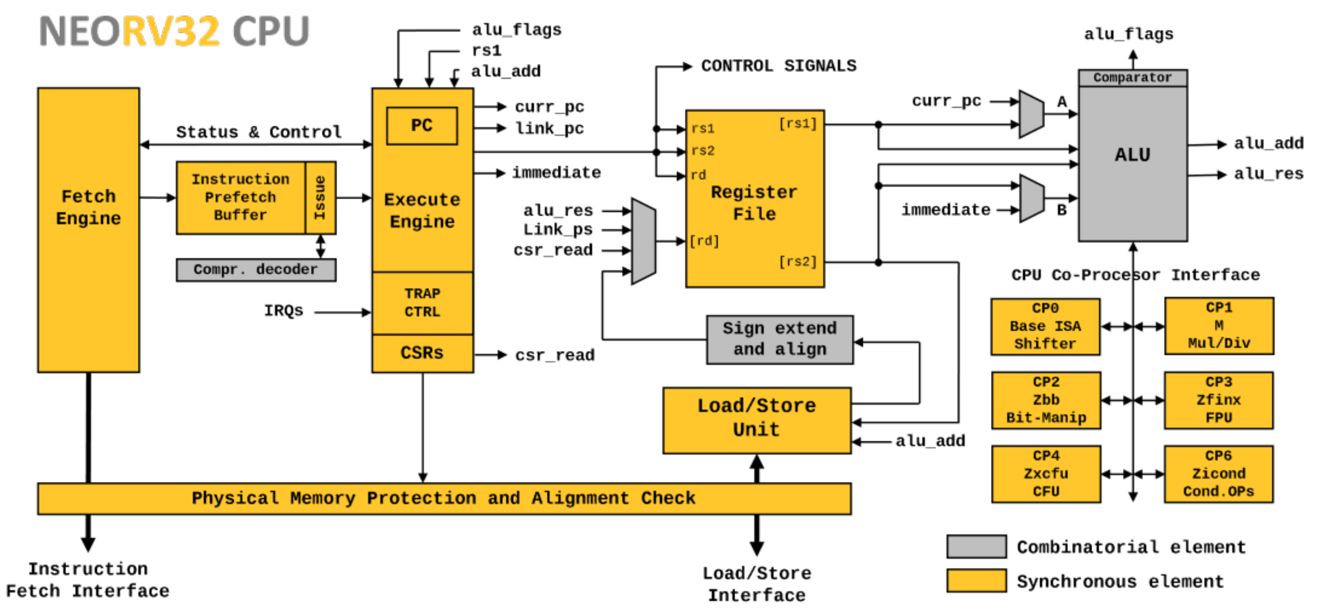
Table 44. NEORV32 CPU-Exclusive Generic List

Name	Type	Description
`HART_ID`	natural	ID of the core (for **`mhartid`** CSR).
`NUM_HARTS`	natural	Total number of cores in the system.
`VENDOR_ID`	suv(31:0)	Vendor identification (for **`mvendorid`** CSR).
`BOOT_ADDR`	suv(31:0)	CPU reset address. See section Address Space.
`DEBUG_PARK_ADDR`	suv(31:0)	"Park loop" entry address for the On-Chip Debugger (OCD), has to be 4-byte aligned.
`DEBUG_EXC_ADDR`	suv(31:0)	"Exception" entry address for the On-Chip Debugger (OCD), has to be 4-byte aligned.
`ICC_EN`	boolean	Implement Inter-Core Communication (ICC) module. Automatically enabled for the SMP Dual-Core Configuration.
`RISCV_ISA_Sdext`	boolean	Implement RISC-V-compatible "debug" CPU operation mode required for the On-Chip Debugger (OCD).
`RISCV_ISA_Sdtrig`	boolean	Implement RISC-V-compatible trigger module. See section On-Chip Debugger (OCD).
`RISCV_ISA_Smpmp`	boolean	Implement RISC-V-compatible physical memory protection (PMP). See section `Smpmp` ISA Extension.

Tuning Option Generics

Additional generics that are related to certain *tuning options* are listed in section CPU Tuning Options.

3.4. Architecture



The CPU implements a pipelined multi-cycle architecture: each instruction is executed as a series of consecutive micro-operations. In order to increase performance, the CPU's front-end (instruction fetch) and back-end (instruction execution) are de-couples via a FIFO (the instruction prefetch buffer. Thus, the front-end can already fetch new instructions while the back-end is still processing the previously-fetched instructions.

Basically, the CPU's micro architecture is somewhere between a classical pipelined architecture, where each stage requires exactly one processing cycle (if not stalled) and a classical multi-cycle architecture, which executes every single instruction (*including* fetch) in a series of consecutive micro-operations. The combination of these two design paradigms allows an increased instruction execution in contrast to a pure multi-cycle approach (due to overlapping operation of fetch and execute) at a reduced hardware footprint (due to the multi-cycle concept).

As a Von-Neumann machine, the CPU provides independent interfaces for instruction fetch and data access. However, these two bus interfaces are merged into a single processor-internal bus via a prioritizing bus switch (data accesses have higher priority). Hence, *all* memory addresses including peripheral devices are mapped to a single unified 32-bit Address Space.

Linear/In-Order Execution Only

The CPU does not perform any speculative/out-of-order operations at all. Hence, it is not vulnerable to security issues caused by speculative execution (like Spectre or Meltdown).

3.4.1. CPU Register File

The data register file contains the general purpose architecture registers `x0` to `x31`. For the `rv32e` ISA only the lower 16 registers are implemented. Register zero (`x0`/`zero`) always read as zero and any write access to it has no effect. Up to four individual synchronous read ports allow to fetch up to 4 register operands at once. The write and read accesses are mutually exclusive as they happen in separate cycles. Hence, there is no need to consider things like "read-during-write" behavior.

Memory Tuning Options

The physical implementation of the register file's memory core can be tuned for certain design goals like area or throughput. See section CPU Tuning Options for more information.

Implementation of the `zero` Register within FPGA Block RAM

Register `zero` is also mapped to a *physical memory location* within the register file's block RAM. By this, there is no need to add a further multiplexer to "insert" zero if reading from register `zero` reducing logic requirements and shortening the critical path. However, this also requires that the physical storage bits of register `zero` are explicitly initialized (set to zero) by the hardware. This is done transparently by the CPU control requiring no additional processing overhead.

Block RAM Ports

The default register file configuration uses two access ports: a read-only port for reading register `rs2` (second source operand) and a read/write port for reading register `rs1` (first source operand) and for writing processing results to register `rd` (destination register). Hence, a simple dual-port RAM can be used to implement the entire register file. From a functional point of view, read and write accesses to the register file do never occur in the same clock cycle, so no bypass logic is required at all.

3.4.2. CPU Arithmetic Logic Unit

The arithmetic/logic unit (ALU) is used for actual data processing as well as generating memory and branch addresses. All "simple" `I` ISA Extension computational instructions (like `add` and `or`) are implemented as plain combinatorial logic requiring only a single cycle to complete. More sophisticated instructions like shift operations or multiplications are processed by so-called "ALU co-processors".

The co-processors are implemented as iterative units that require several cycles to complete processing. Besides the base ISA's shift instructions, the co-processors are used to implement all further processing-based ISA extensions (e.g. `M` ISA Extension and `B` ISA Extension).

Multi-Cycle Execution Monitor

The CPU control will raise an illegal instruction exception if a multi-cycle functional unit (like the Custom Functions Unit (CFU)) does not complete processing in a bound amount of time (configured via the package's `monitor_mc_tmo_c` constant; default = 512 clock cycles).

3.4.3. CPU Bus Unit

The bus unit takes care of handling data memory accesses via load and store instructions. It handles data adjustment when accessing sub-word data quantities (16-bit or 8-bit) and performs sign-extension for singed load operations. The bus unit also includes the optional `Smpmp` ISA Extension that performs permission checks for all data and instruction accesses.

A list of the bus interface signals and a detailed description of the protocol can be found in section Bus Interface. All bus interface signals are driven/buffered by registers; so even a complex SoC interconnection bus network will not effect maximal operation frequency.

Unaligned Accesses

The CPU does not support a hardware-based handling of unaligned memory accesses! Any unaligned access will raise a bus load/store unaligned address exception. The exception handler can be used to *emulate* unaligned memory accesses in software. See the NEORV32 Runtime Environment's Application Context Handling section for more information.

3.4.4. CPU Control Unit

The CPU control unit is responsible for generating all the control signals for the different CPU modules. The control unit is split into a "front-end" and a "back-end".

Front-End

The front-end is responsible for fetching instructions in chunks of 32-bits. This can be a single aligned 32-bit instruction, two aligned 16-bit instructions or a mixture of those. The instructions including control and exception information are stored to a FIFO queue - the instruction prefetch buffer (IPB). This FIFO has a depth of two entries by default but can be customized via the `ipb_depth_c` VHDL package constant.

The FIFO allows the front-end to do "speculative" instruction fetches, as it keeps fetching the next consecutive instruction all the time. This also allows to decouple front-end (instruction fetch) and back-end (instruction execution) so both modules can operate in parallel to increase performance. However, all potential side effects that are caused by this "speculative" instruction fetch are already handled by the CPU front-end ensuring a defined execution stage while preventing security side attacks.

Back-End

Instruction data from the instruction prefetch buffer is decompressed (if the `C` ISA extension is enabled) and sent to the CPU back-end for actual execution. Execution is conducted by a state-machine that controls all of the CPU modules. The back-end also includes the Control and Status Registers (CSRs) as well as the trap controller.

3.4.5. CPU Tuning Options

The top module provides several tuning options to optimize the CPU for a specific goal. Note that these configuration options have no impact on the actual functionality (e.g. ISA compatibility).

Software Tuning Options Discovery

Software can check for configured tuning options via specific flags in the `mxisa` CSR.

`CPU_CLOCK_GATING_EN`

Name	Clock gating
Type	`boolean`
Default	`false`, disabled
Description	When **enabled** the CPU's primary clock is switched off when the CPU enters Sleep Mode. See CPU Clock Gating. When **disabled** the CPU clock system is implemented as single always-on clock domain.

`CPU_FAST_MUL_EN`

Name	Fast multiplication
Type	`boolean`
Default	`false`, disabled
Description	When **enabled** the `M`/`Zmmul` extension's multiplier is implemented as "plain multiplication" allowing the synthesis tool to infer DSP blocks / multiplication primitives. Multiplication operations only require a few cycles due to the DSP-internal register stages. The execution time is time-independent of the provided operands. When **disabled** the `M`/`Zmmul` extension's multiplier is implemented as bit-serial multiplier that computes one result bit in every cycle. Multiplication operations only requires at least 32 cycles but the entire execution time is still time-independent of the provided operands.

`CPU_FAST_SHIFT_EN`

Name	Fast bit shifting
Type	`boolean`
Default	`false`, disabled

Copyright by Stephan Nolting. All rights reserved.

Description	When **enabled** the ALU's shifter unit is implemented as full-parallel barrel shifter that is capable of shifting a data word by an arbitrary number of positions within a single cycle. Hence, the execution time of any base-ISA shift operation is independent of the provided operands. Note that the barrel shifter requires a lot of hardware resources and might also increase the core's critical path. When **disabled** the ALU's shifter unit is implemented as bit-serial shifter that can shift the input data only by one position per cycle. Hence, several cycles might be required to complete any base-ISA shift-related operations. Therefore, the execution time of the serial approach is **not** time-independent of the provided operands. However, the serial approach requires only a few hardware resources and does not impact the critical path.

`CPU_RF_HW_RST_EN`

Name	Register file hardware reset
Type	`boolean`
Default	`false`, disabled
Description	When **enabled** the CPU register file is implemented using single flip flops that provide a full hardware reset. The register file is reset to all-zero after each hardware reset. Note that this options requires a lot of flip flops and LUTs to build the register file. However, timing might be optimized as there is no need to route to far blockRAM resources. When **disabled** the CPU register file is implemented in a way to allow synthesis to infer memory primitives like blockRAM. Note that these primitives do not provide any kind of hardware reset. Hence, the data content is undefined after reset.

3.4.6. Sleep Mode

The NEORV32 CPU provides a single sleep mode that can be entered to power-down the core reducing dynamic power consumption. Sleep mode is entered by executing the RISC-V `wfi` ("wait for interrupt") instruction.

Execution Details

The `wfi` instruction will raise an illegal instruction exception when executed in user-mode if `TW` in **`mstatus`** is set. When executed in debug-mode or during single-stepping `wfi` will behave as simple `nop` without entering sleep mode.

After executing the `wfi` instruction the `sleep` signal of the CPU's request buses (Bus Interface will become set as soon as the CPU has fully halted:

There is no enabled interrupt being pending.

CPU-external modules like memories, timers and peripheral interfaces are not affected by this.

Furthermore, the CPU will continue to buffer/enqueue incoming interrupts. The CPU will leave sleep mode as soon as any *enabled* interrupt (via `mie`) source becomes *pending* or if a debug session is started.

3.4.7. CPU Clock Gating

The single clock domain of the CPU core can be split into an always-on clock domain and a switchable clock domain. The switchable clock domain can be deactivated to further reduce reduce dynamic power consumption. CPU-external modules like timers, interfaces and memories are not affected by the clock gating.

The splitting into two clock domain is enabled by the `CPU_CLOCK_GATING_EN` generic (Processor Top Entity - Generics / CPU Tuning Options). When enabled, a generic clock switching gate is added to decouple the switchable clock from the always-on clock domain. Whenever the CPU enters Sleep Mode the switchable clock domain is shut down.

Clock Switch Hardware

By default, a generic clock switch is used (`rtl/core/neorv32_clockgate.vhd`). Especially for FPGA setups it is highly recommended to replace this default module by a technology-specific primitive or macro wrapper to improve synthesis results (clock skew, global clock tree usage, etc.).

3.4.8. Full Virtualization

Just like the RISC-V ISA, the NEORV32 aims to provide *maximum virtualization* capabilities on CPU and SoC level to allow a high standard of **execution safety**. The CPU supports **all** traps specified by the official RISC-V specifications. Thus, the CPU provides defined hardware fall-backs via traps for any expected and unexpected situations (e.g. executing a malformed or not supported instruction or accessing a non-allocated memory address). For any kind of trap the core is always in a defined and fully synchronized state throughout the whole system (i.e. there are no out-of-order operations that might have to be reverted). This allows a defined and predictable execution behavior at any time improving overall execution safety.

3.5. Bus Interface

The NEORV32 CPU provides separated instruction fetch and data access interfaces making it a **Harvard Architecture**: the instruction fetch interface (`i_bus_*` signals) is used for fetching instructions and the data access interface (`d_bus_*` signals) is used to access data via load and store operations. Each of these interfaces can access an address space of up to 2^{32} bytes (4GB).

The bus interface uses two custom interface types: `bus_req_t` is used to propagate the bus access requests downstream from a host to a device. These signals are driven by the request-issuing device (i.e. the CPU core). Vice versa, `bus_rsp_t` is used to return the bus response upstream from a device back to the host and is driven by the accessed device or bus system (i.e. a processor-internal memory or IO device).

The signals of the request bus are split in to two categories: *in-band* signals and *out-of-band* signals. In-band signals always belong to a certain bus transaction and are only valid between `stb` being set and the according response (`err` or `ack`). being set. In contrast, the out-of-band signals are not associated with any bus transaction and are always valid when set.

Table 45. Bus Interface - Request Bus (`bus_req_t`)

Signal	Width	Description
		In-Band Signals
`addr`	32	Access address (byte addressing)
`data`	32	Write data
`ben`	4	Byte-enable for each byte in `data`
`stb`	1	Request trigger ("strobe", single-shot)
`rw`	1	Access direction (`0` = read, `1` = write)
`src`	1	Access source (`0` = instruction fetch, `1` = load/store)
`priv`	1	Set if privileged (M-mode) access
`amo`	1	Set if current access is an atomic memory operation (Atomic Memory Access)
`amoop`	4	Type of atomic memory operation (Atomic Memory Access)
		Out-Of-Band Signals
`fence`	1	Data/instruction fence request; single-shot
`sleep`	1	Set if ALL upstream devices are in Sleep Mode
`debug`	1	Set if the upstream device is in debug-mode

Table 46. Bus Interface - Response Bus (`bus_rsp_t`)

Signal	Width	Description
`data`	32	Read data (single-shot)

Signal	Width	Description
`ack`	1	Transfer acknowledge / success (single-shot)
`err`	1	Transfer error / fail (single-shot)

3.5.1. Bus Interface Protocol

Transactions are triggered entirely by the request bus. A new bus request is initiated by setting the *strobe* signal `stb` high for exactly one cycle. All remaining signals of the bus are set together with `stb` and will remain unchanged until the transaction is completed.

The transaction is completed when the accessed device returns a response via the response interface: `ack` is high for exactly one cycle if the transaction was completed successfully. `err` is high for exactly one cycle if the transaction failed to complete. These two signals are mutually exclusive. In case of a read access the read data is returned together with the `ack` signal. Otherwise, the return data signal is kept at all-zero allowing wired-or interconnection of all response buses.

The figure below shows three exemplary bus accesses:

1. A read access to address `A_addr` returning `rdata` after several cycles (slow response; `ACK` arrives after several cycles).
2. A write access to address `B_addr` writing `wdata` (fastest response; `ACK` arrives right in the next cycle).
3. A failing read access to address `C_addr` (slow response; `ERR` arrives after several cycles).

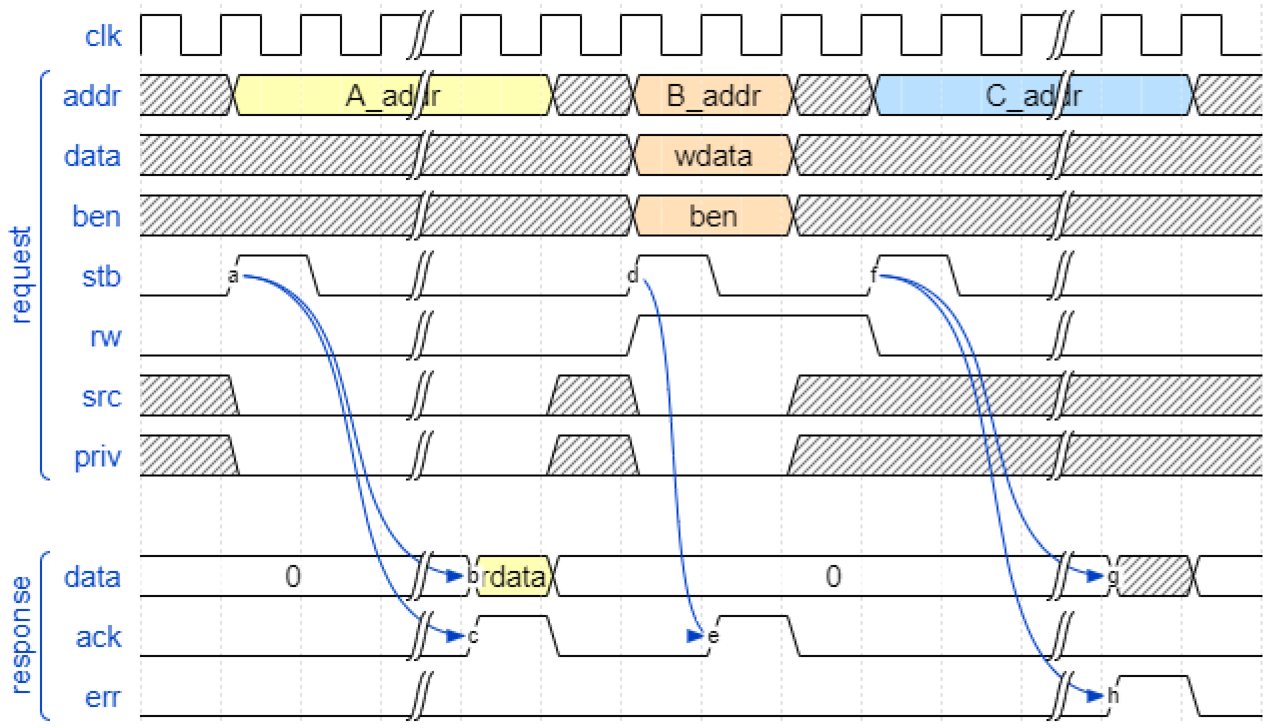


Figure 9. Three Exemplary Bus Transactions (showing only in-band signals)

Adding Register Stages

Arbitrary pipeline stages can be added to the request and response buses at any point to relax timing (at the cost of additional latency). However, *all* bus signals (request and response) need to be registered.

3.5.2. Atomic Memory Access

The `Zaamo` ISA Extension adds atomic read-modify-write memory operations. Since the Bus Interface Protocol only supports read-or-write operations, the atomic memory requests are handled by a dedicated module of the bus infrastructure - the Atomic Memory Operations Controller.

For the CPU, the atomic memory accesses are handled as plain "load" operation but with the `amo` signal set and also providing write data (see Bus Interface). The `amoop` signal defines the actual atomic processing operation:

Table 47. AMO Operation Type Encoding

`bus_req_t.amoop`	**Description**
`-000`	swap
`-001`	unsigned add
`-010`	logical xor
`-011`	logical and
`-100`	logical or
`0110`	unsigned minimum
`0111`	unsigned maximum
`1110`	signed minimum
`1111`	signed maximum

Cache Coherency

Atomic operations **always bypass** the CPU caches using direct/uncached accesses. Care must be taken to maintain data Cache Coherency.

3.6. Instruction Sets and Extensions

The NEORV32 CPU provides several optional RISC-V-compliant and custom/user-defined ISA extensions. The extensions can be enabled/configured via the according Processor Top Entity - Generics. This chapter gives a brief overview of all available ISA extensions.

Table 48. NEORV32 Instruction Set Extensions

Name	Description	Enabled by Generic
`B`	Bit manipulation instructions	*Implicitly* enabled
`C`	Compressed (16-bit) instructions	`RISCV_ISA_C`
`E`	Embedded CPU extension (reduced register file size)	`RISCV_ISA_E`
`I`	Integer base ISA	Enabled if `RISCV_ISA_E` is **not** enabled
`M`	Integer multiplication and division instructions	`RISCV_ISA_M`
`U`	Less-privileged *user* mode extension	`RISCV_ISA_U`
`X`	Platform-specific / NEORV32-specific extension	Always enabled
`Zaamo`	Atomic memory operations	`RISCV_ISA_Zaamo`
`Zba`	Shifted-add bit manipulation instructions	`RISCV_ISA_Zba`
`Zbb`	Basic bit manipulation instructions	`RISCV_ISA_Zbb`
`Zbkb`	Scalar cryptographic bit manipulation instructions	`RISCV_ISA_Zbkb`
`Zbkc`	Scalar cryptographic carry-less multiplication instructions	`RISCV_ISA_Zbkc`
`Zbkx`	Scalar cryptographic crossbar permutation instructions	`RISCV_ISA_Zbkx`
`Zbs`	Single-bit bit manipulation instructions	`RISCV_ISA_Zbs`
`Zfinx`	Floating-point instructions using integer registers	`RISCV_ISA_Zfinx`
`Zifencei`	Instruction stream synchronization instruction	Always enabled
`Zicntr`	Base counters extension	`RISCV_ISA_Zicntr`
`Zicond`	Integer conditional operations	`RISCV_ISA_Zicond`
`Zicsr`	Control and status register access instructions	Always enabled
`Zihpm`	Hardware performance monitors extension	`RISCV_ISA_Zihpm`
`Zkn`	Scalar cryptographic NIST algorithm suite	*Implicitly* enabled
`Zknd`	Scalar cryptographic NIST AES decryption instructions	`RISCV_ISA_Zknd`

Name	Description	Enabled by Generic
`Zkne`	Scalar cryptographic NIST AES encryption instructions	`RISCV_ISA_Zkne`
`Zknh`	Scalar cryptographic NIST hash function instructions	`RISCV_ISA_Zknh`
`Zkt`	Data independent execution time (of cryptographic operations)	*Implicitly* enabled
`Zks`	Scalar cryptographic ShangMi algorithm suite	*Implicitly* enabled
`Zksed`	Scalar cryptographic ShangMi block cypher instructions	`RISCV_ISA_Zksed`
`Zksh`	Scalar cryptographic ShangMi hash instructions	`RISCV_ISA_Zksh`
`Zmmul`	Integer multiplication-only instructions	`RISCV_ISA_Zmmul`
`Zcfu`	Custom / user-defined instructions	`RISCV_ISA_Zxcfu`
`Smpmp`	Physical memory protection (PMP) extension	`RISCV_ISA_Smpmp`
`Sdext`	External debug support extension	`OCD_EN`
`Sdtrig`	Trigger module extension	`OCD_EN`

RISC-V ISA Specification

For more information regarding the RISC-V ISA extensions please refer to the "RISC-V Instruction Set Manual - Volume I: Unprivileged ISA" and "The RISC-V Instruction Set Manual Volume II: Privileged Architecture". A copy of these documents can be found in the projects `docs/references` folder.

Discovering ISA Extensions

Software can discover available ISA extensions via the **`misa`** and **`mxisa`** CSRs or by executing an instruction and checking for an illegal instruction exception (i.e. Full Virtualization).

Instruction Cycles

This chapter shows the CPI values (cycles per instruction) for each individual instruction/type. Note that values reflect *optimal conditions* (i.e. no additional memory delay, no cache misses, no pipeline waits, etc.). To benchmark a certain processor configuration for its setup-specific CPI value please refer to the `sw/example/performance_tests` test programs.

3.6.1. `B` ISA Extension

The `B` ISA extension adds instructions for bit-manipulation operations. This ISA extension cannot be enabled by a specific generic. Instead, it is enabled if a specific set of bit-manipulation sub-extensions are enabled.

The B extension is shorthand for the following set of other extensions:

- Zba ISA Extension - Address-generation / shifted-add instructions.
- Zbb ISA Extension - Basic bit manipulation instructions.
- Zbs ISA Extension - Single-bit operations.

A processor configuration which implements B must implement all of the above extensions.

3.6.2. C ISA Extension

The "compressed" ISA extension provides 16-bit encodings of commonly used instructions to reduce code space size.

Table 49. Instructions and Timing

Class	Instructions	Execution cycles
ALU	`c.addi4spn` `c.nop` `c.add[i]` `c.li` `c.addi16sp` `c.lui` `c.and[i]` `c.sub` `c.xor` `c.or` `c.mv`	2
ALU	`c.srli` `c.srai` `c.slli`	3 + 1..32; `CPU_FAST_SHIFT_EN`: 4
Branches	`c.beqz` `c.bnez`	taken: 6; not taken: 3
Jumps / calls	`c.jal[r]` `c.j` `c.jr`	6
Memory access	`c.lw` `c.sw` `c.lwsp` `c.swsp`	4
System	`c.break`	3

3.6.3. E ISA Extension

The "embedded" ISA extensions reduces the size of the general purpose register file from 32 entries to 16 entries to shrink hardware size. It provides the same instructions as the the base I ISA extensions.

Alternative MABI

Due to the reduced register file size an alternate toolchain ABI (`ilp32e*`) is required.

3.6.4. I ISA Extension

The I ISA extensions is the base RISC-V integer ISA that is always enabled.

Table 50. Instructions and Timing

Class	Instructions	Execution cycles
ALU	`add[i]` `slt[i]` `slt[i]u` `xor[i]` `or[i]` `and[i]` `sub` `lui` `auipc`	2
No-operation	“nop”	2

Class	Instructions	Execution cycles
ALU shifts	`sll[i]` `srl[i]` `sra[i]`	3 + 1..32; `CPU_FAST_SHIFT_EN`: 4
Branches	`beq` `bne` `blt` `bge` `bltu` `bgeu`	taken: 6; not taken: 3
Jump/call	`jal[r]`	6
Load/store	`lb` `lh` `lw` `lbu` `lhu` `sb` `sh` `sw`	5
System	`ecall` `ebreak`	3
Data fence	`fence`	5
System	`wfi`	3
System	`mret`	5
Illegal inst.	-	3

`fence` Instruction

Analogous to the `fence.i` instruction (`Zifencei` ISA Extension) the `fence` instruction triggers a data cache synchronization operation. See section Cache Coherency for more information. Furthermore, the `fence` instruction word's *predecessor* and *successor* bits (used for memory ordering) are not evaluated by the hardware at all.

`wfi` Instruction

The `wfi` instruction is used to enter Sleep Mode. Executing the `wfi` instruction in user-mode will raise an illegal instruction exception if the `TW` bit of **`mstatus`** is set.

Shifter Tuning Options

The physical implementation of the bit-shifter can be tuned for certain design goals like area or throughput. See section CPU Tuning Options for more information.

3.6.5. `M` ISA Extension

Hardware-accelerated integer multiplication and division operations are available via the RISC-V `M` ISA extension. This ISA extension is implemented as multi-cycle ALU co-process (`rtl/core/neorv32_cpu_cp_muldiv.vhd`).

Table 51. Instructions and Timing

Class	Instructions	Execution cycles
Multiplication	`mul` `mulh` `mulhsu` `mulhu`	36; `CPU_FAST_MUL_EN`: 4
Division	`div` `divu` `rem` `remu`	36

Multiplication Tuning Options

The physical implementation of the multiplier can be tuned for certain design goals like area or throughput. See section CPU Tuning Options for more information.

3.6.6. `U` ISA Extension

In addition to the highest-privileged machine-mode, the user-mode ISA extensions adds a second **less-privileged** operation mode. Code executed in user-mode has reduced CSR access rights. Furthermore, user-mode accesses to the address space (like peripheral/IO devices) can be constrained via the physical memory protection. Any kind of privilege rights violation will raise an exception to allow Full Virtualization.

3.6.7. `X` ISA Extension

The NEORV32-specific ISA extensions `X` is always enabled. The most important points of the NEORV32-specific extensions are: * The CPU provides 16 *fast interrupt* interrupts (`FIRQ`), which are controlled via custom bits in the **`mie`** and **`mip`** CSRs. These extensions are mapped to CSR bits, that are available for custom use according to the RISC-V specs. Also, custom trap codes for **`mcause`** are implemented. * All undefined/unimplemented/malformed/illegal instructions do raise an illegal instruction exception (see Full Virtualization). * There are NEORV32-Specific CSRs.

3.6.8. `Zaamo` ISA Extension

The `Zaamo` ISA extension is a sub-extension of the RISC-V `A` ISA extension and compromises instructions for read-modify-write Atomic Memory Access operations. It is enabled by the top's `RISCV_ISA_Zaamo` generic.

Table 52. Instructions and Timing

Class	Instructions	Execution cycles
Atomic memory operations	`amoswap.w` `amoadd.w` `amoand.w` `amoor.w` `amoxor.w` `amomax[u].w` `amomin[u].w`	5 + 2 * *memory_latency*

`aq` and `rl` Bits

The instruction word's `aq` and `lr` memory ordering bits are not evaluated by the hardware at all.

3.6.9. `Zifencei` ISA Extension

The `Zifencei` CPU extension allows manual synchronization of the instruction stream. This extension is always enabled.

Analogous to the `fence` instruction the `fence.i` instruction triggers an instruction cache synchronization operation. See section Cache Coherency for more information.

Table 53. Instructions and Timing

Class	Instructions	Execution cycles
Instruction fence	`fence.i`	5

3.6.10. `Zfinx` ISA Extension

The `Zfinx` floating-point extension is an *alternative* of the standard `F` floating-point ISA extension. It also uses the integer register file `x` to store and operate on floating-point data instead of a dedicated floating-point register file. Thus, the `Zfinx` extension requires less hardware resources and features faster context changes. This also implies that there are NO dedicated `f` register file-related load/store or move instructions. The `Zfinx` extension'S floating-point unit is controlled via dedicated Floating-Point CSRs. This ISA extension is implemented as multi-cycle ALU co-process (`rtl/core/neorv32_cpu_cp_fpu.vhd`).

Fused / Multiply-Add Instructions

Fused multiply-add instructions `f[n]m[add/sub].s` are not supported. A special GCC switch is used to prevent the compiler from emitting contracted/fused floating-point operations (see Default Compiler Flags).

Division and Squarer Root Instructions

Division `fdiv.s` and square root `fsqrt.s` instructions are not supported yet.

Subnormal Number

Subnormal numbers ("de-normalized" numbers, i.e. exponent = 0) are not supported by the NEORV32 FPU. Subnormal numbers are *flushed to zero* setting them to +/- 0 before being processed by **any** FPU operation. If a computational instruction generates a subnormal result it is also flushed to zero during normalization.

Table 54. Instructions and Timing

Class	Instructions	Execution cycles
Artihmetic	`fadd.s`	110
Artihmetic	`fsub.s`	112
Artihmetic	`fmul.s`	22
Compare	`fmin.s` `fmax.s` `feq.s` `flt.s` `fle.s`	13
Conversion	`fcvt.w.s` `fcvt.wu.s` `fcvt.s.w` `fcvt.s.wu`	48
Misc	`fsgnj.s` `fsgnjn.s` `fsgnjx.s` `fclass.s`	12

3.6.11. `Zicntr` ISA Extension

The `Zicntr` ISA extension adds the basic **`cycle[h]`**, **`mcycle[h]`**, **`instret[h]`** and **`minstret[h]`** counter

CSRs. Section (Machine) Counter and Timer CSRs shows a list of all `Zicntr`-related CSRs.

Time CSRs

The user-mode `time[h]` CSRs are **not implemented**. Any access will trap allowing the trap handler to retrieve system time from the Core Local Interruptor (CLINT).

Mandatory Extension

This extensions is stated as *mandatory* by the RISC-V spec. However, area-constrained setups may remove support for these counters.

Constrained Access

User-level access to the counter CSRs can be constrained by the **`mcounteren`** CSR.

3.6.12. `Zicond` ISA Extension

The `Zicond` ISA extension adds integer conditional move primitives that allow to implement branch-less control flows. It is enabled by the top's `RISCV_ISA_Zicond` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_cond.vhd`).

Table 55. Instructions and Timing

Class	Instructions	Execution cycles
Conditional	`czero.eqz` `czero.nez`	3

3.6.13. `Zicsr` ISA Extension

This ISA extensions provides instructions for accessing the Control and Status Registers (CSRs) as well as further privileged-architecture extensions. This extension is mandatory and cannot be disabled. Hence, there is no generic for enabling/disabling this ISA extension.

Side-Effects if Destination is Zero-Register

If `rd=x0` for the `csrrw[i]` instructions there will be no actual read access to the according CSR. However, access privileges are still enforced so these instruction variants *do* cause side-effects (the RISC-V spec. state that these combinations "shall" not cause any side-effects).

Table 56. Instructions and Timing

Class	Instructions	Execution cycles
System	`csrrw[i]` `csrrs[i]` `csrrc[i]`	3

3.6.14. `Zihpm` ISA Extension

In additions to the base counters the NEORV32 CPU provides up to 13 hardware performance monitors (HPM 3..15), which can be used to benchmark applications. Each HPM consists of an N-bit wide counter (split in a high-word 32-bit CSR and a low-word 32-bit CSR), where N is defined via the top's `HPM_CNT_WIDTH` generic and a corresponding event configuration CSR.

The event configuration CSR defines the architectural events that lead to an increment of the associated HPM counter. See section Hardware Performance Monitors (HPM) CSRs for a list of all HPM-related CSRs and event configurations.

Machine-Mode HPMs Only

Note that only the machine-mode hardware performance counter CSR are available (`mhpmcounter*[h]`). Accessing any user-mode HPM CSR (`hpmcounter*[h]`) will raise an illegal instruction exception.

Increment Inhibit

The event-driven increment of the HPMs can be deactivated individually via the **`mcountinhibit`** CSR.

3.6.15. `Zba` ISA Extension

The `Zba` sub-extension is part of the *RISC-V bit manipulation* ISA specification (`B` ISA Extension) and adds shifted-add / address-generation instructions. It is enabled by the top's `RISCV_ISA_Zba` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_bitmanip.vhd`).

Table 57. Instructions and Timing

Class	Instructions	Execution cycles
Shifted-add	`sh1add` `sh2add` `sh3add`	4

3.6.16. `Zbb` ISA Extension

The `Zbb` sub-extension is part of the *RISC-V bit manipulation* ISA specification (`B` ISA Extension) and adds the basic bit manipulation instructions. It is enabled by the top's `RISCV_ISA_Zbb` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_bitmanip.vhd`).

Table 58. Instructions and Timing

Class	Instructions	Execution cycles
Logic with negate	`andn` `orn` `xnor`	4
Count leading/trailing zeros	`clz` `ctz`	6 + 1..32; `CPU_FAST_SHIFT_EN`: 4
Count population	`cpop`	6 + 32; `CPU_FAST_SHIFT_EN`: 4
Integer maximum/minimum	`min[u]` `max[u]`	4
Sign/zero extension	`sext.b` `sext.h` `zext`	4

Class	Instructions	Execution cycles
Bitwise rotation	`rol` `ror[i]`	6 + *shift_amount*; `CPU_FAST_SHIFT_EN`: 4
OR-combine	`orc.b`	4
Byte-reverse	`rev8`	4

shifter Tuning Options

The physical implementation of the bit-shifter can be tuned for certain design goals like area or throughput. See section CPU Tuning Options for more information.

3.6.17. `Zbs` ISA Extension

The `Zbs` sub-extension is part of the *RISC-V bit manipulation* ISA specification (`B` ISA Extension) and adds single-bit operations. It is enabled by the top's `RISCV_ISA_Zbs` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_bitmanip.vhd`).

Table 59. Instructions and Timing

Single-bit	`sbset[i]` `sbclr[i]` `sbinv[i]` `sbext[i]`	**4**

3.6.18. `Zbkb` ISA Extension

The `Zbkb` sub-extension is part of the *RISC-V scalar cryptography* ISA specification and extends the *RISC-V bit manipulation* ISA extension with additional instructions. It is enabled by the top's `RISCV_ISA_Zbkb` generic. Note that enabling this extension will also enable the `Zbb` basic bit-manipulation ISA extension (which is extended by `Zknb`). This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_bitmanip.vhd`).

Table 60. Instructions and Timing (in addition to `Zbb`)

Class	Instructions	Execution cycles
Packing	`pack` `packh`	4
Interleaving	`zip` `unzip`	4
Byte-wise bit reversal	`brev8`	4

3.6.19. `Zbkc` ISA Extension

The `Zbkc` sub-extension is part of the *RISC-V scalar cryptography* ISA extension and adds carry-less multiplication instruction. ISA extension with additional instructions. It is enabled by the top's `RISCV_ISA_Zbkc` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_bitmanip.vhd`).

Table 61. Instructions and Timing

Class	Instructions	Execution cycles
Carry-less multiply	`clmul` `clmulh`	6 + 32

3.6.20. `Zbkx` ISA Extension

The `Zbkx` sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds crossbar permutation instructions. It is enabled by the top's `RISCV_ISA_Zbkx` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_crypto.vhd`).

Table 62. Instructions and Timing

Class	Instructions	Execution cycles
Crossbar permutation	`xperm8` `xperm4`	4

3.6.21. `Zkn` ISA Extension

The `Zkn` ISA extension is part of the *RISC-V scalar cryptography* ISA specification and defines the "NIST algorithm suite". This ISA extension cannot be enabled by a specific generic. Instead, it is enabled if a specific set of cryptography-related sub-extensions is enabled.

The `Zkn` extension is shorthand for the following set of other extensions:

- `Zbkb` ISA Extension - Bit manipulation instructions for cryptography.
- `Zbkc` ISA Extension - Carry-less multiply instructions.
- `Zbkx` ISA Extension - Cross-bar permutation instructions.
- `Zkne` ISA Extension - AES encryption instructions.
- `Zknd` ISA Extension - AES decryption instructions.
- `Zknh` ISA Extension - SHA2 hash function instructions.

A processor configuration which implements `Zkn` must implement all of the above extensions.

3.6.22. `Zknd` ISA Extension

The `Zknd` sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds NIST AES decryption instructions. It is enabled by the top's `RISCV_ISA_Zknd` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_crypto.vhd`).

Table 63. Instructions and Timing

Class	Instructions	Execution cycles
AES decryption	`aes32dsi` `aes32dsmi`	6

3.6.23. Zkne ISA Extension

The `Zkne` sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds NIST AES encryption instructions. It is enabled by the top's `RISCV_ISA_Zkne` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_crypto.vhd`).

Table 64. Instructions and Timing

Class	Instructions	Execution cycles
AES decryption	`aes32esi` `aes32esmi`	6

3.6.24. Zknh ISA Extension

The `Zknh` sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds NIST hash function instructions. It is enabled by the top's `RISCV_ISA_Zknh` generic. This ISA extension is implemented as multi-cycle ALU co-processor (`rtl/core/neorv32_cpu_cp_crypto.vhd`).

Table 65. Instructions and Timing

Class	Instructions	Execution cycles
sha256	`sha256sig0` `sha256sig1` `sha256sum0` `sha256sum1`	4
sha512	`sha512sig0h` `sha512sig0l` `sha512sig1h` `sha512sig1l` `sha512sum0r` `sha512sum1r`	4

3.6.25. Zks ISA Extension

The `Zks` ISA extension is part of the *RISC-V scalar cryptography* ISA specification and defines the "ShangMi algorithm suite". This ISA extension cannot be enabled by a specific generic. Instead, it is enabled if a specific set of cryptography-related sub-extensions is enabled.

The `Zks` extension is shorthand for the following set of other extensions:

- `Zbkb` ISA Extension - Bit manipulation instructions for cryptography.
- `Zbkc` ISA Extension - Carry-less multiply instructions.
- `Zbkx` ISA Extension - Cross-bar permutation instructions.
- `Zksed` ISA Extension - SM4 block cipher instructions.
- `Zksh` ISA Extension - SM3 hash function instructions.

A processor configuration which implements `Zks` must implement all of the above extensions.

3.6.26. Zksed ISA Extension

The `Zksed` sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds ShangMi block cypher and key schedule instructions. It is enabled by the top's `RISCV_ISA_Zksed` generic. This ISA extension is implemented as multi-cycle ALU co-processor (rtl/core/neorv32_cpu_cp_crypto.vhd).

Table 66. Instructions and Timing

Class	Instructions	Execution cycles
Block cyphers	sm4ed	6
Key schedule	sm4ks	6

3.6.27. Zksh ISA Extension

The Zksh sub-extension is part of the *RISC-V scalar cryptography* ISA specification and adds ShangMi hash function instructions. It is enabled by the top's RISCV_ISA_Zksh generic. This ISA extension is implemented as multi-cycle ALU co-processor (rtl/core/neorv32_cpu_cp_crypto.vhd).

Table 67. Instructions and Timing

Class	Instructions	Execution cycles
Hash	sm3p0 sm3p1	6

3.6.28. Zkt ISA Extension

The Zkt sub-extension is part of the *RISC-V scalar cryptography* ISA specification and guarantees data independent execution times of cryptographic and cryptography-related instructions. The ISA extension cannot be enabled by a specific generic. Instead, it is enabled implicitly by certain CPU configurations.

The RISC-V Zkt specifications provides a list of instructions that are included within this specification. However, not all instructions are required to be implemented. Rather, every one of these instructions that the core does implement must adhere to the requirements of Zkt.

Table 68. Zkt instruction listing

Parent extension	Instructions	Data independent execution time?
RVI	lui auipc add[i] slt[i][u] xor[i] or[i] and[i] sub	yes
	sll[i] srl[i] sra[i]	yes if CPU_FAST_SHIFT_EN enabled
RVM	mul[h] mulh[s]u	yes
RVC	c.nop c.addi c.lui c.andi c.sub c.xor c.and c.mv c.add	yes
	c.srli c.srai c.slli	yes if CPU_FAST_SHIFT_EN enabled
RVK	aes32ds[m]i aes32es[m]i sha256sig* sha512sig* sha512sum* sm3p0 sm3p1 sm4ed sm4ks	yes

Parent extension	Instructions	Data independent execution time?
RVB	xperm4 xperm8 andn orn xnor pack[h] brev8 rev8	yes
	ror[i] rol	yes if CPU_FAST_SHIFT_EN enabled

3.6.29. Zmmul - ISA Extension

This is a sub-extension of the M ISA Extension ISA extension. It implements only the multiplication operations of the M extensions and is intended for size-constrained setups that require hardware-based integer multiplications but not hardware-based divisions, which will be computed entirely in software. Note that the Zmmul - ISA Extension and M ISA Extension are mutually exclusive.

3.6.30. Zxcfu ISA Extension

The Zxcfu presents a NEORV32-specific ISA extension. It adds the Custom Functions Unit (CFU) to the CPU core, which allows to add custom RISC-V instructions to the processor core. For detailed information regarding the CFU, its hardware and the according software interface see section Custom Functions Unit (CFU).

Software can utilize the custom instructions by using *intrinsics*, which are basically inline assembly functions that behave like regular C functions but that evaluate to a single custom instruction word (no calling overhead at all).

CFU Execution Time

The actual CFU execution time depends on the logic being implemented. The CPU architecture requires a minimal execution time of 3 cycles (purely combinatorial CFU operation) and automatically terminates execution after 512 cycles if the CFU does not complete operation within this time window.

Table 69. Instructions and Timing

Class	Instructions	Execution cycles
Custom instructions	Instruction words with custom-0 or custom-1 opcode	3 … 3+512

3.6.31. Smpmp ISA Extension

The NEORV32 physical memory protection (PMP) provides an elementary memory protection mechanism that can be used to configure read/write(execute permission of arbitrary memory regions. In general, the PMP can **grant permissions to user mode**, which by default has none, and can **revoke permissions from M-mode**, which by default has full permissions. The NEORV32 PMP is fully compatible to the RISC-V Privileged Architecture Specifications and is configured via several CSRs (Machine Physical Memory Protection CSRs). Several Processor Top Entity - Generics are provided to adjust the CPU's PMP capabilities according to the application requirements (pre-synthesis):

1. `PMP_NUM_REGIONS` defines the number of implemented PMP regions (0..16); setting this generic to zero will result in absolutely no PMP logic being implemented
2. `PMP_MIN_GRANULARITY` defines the minimal granularity of each region (has to be a power of 2, minimal granularity = 4 bytes); note that a smaller granularity will lead to wider comparators and thus, to higher area footprint and longer critical path
3. `PMP_TOR_MODE_EN` controls the implementation of the top-of-region (TOR) mode (default = true); disabling this mode will reduce area footprint
4. `PMP_NAP_MODE_EN` controls the implementation of the naturally-aligned-power-of-two (NA4 and NAPOT) modes (default = true); disabling this mode will reduce area footprint and critical path length

PMP Permissions when in Debug Mode

When in debug-mode all PMP rules are bypassed/ignored granting the debugger maximum access permissions.

PMP Time-Multiplex

Instructions are executed in a multi-cycle manner. Hence, data access (load/store) and instruction fetch cannot occur at the same time. Therefore, the PMP hardware uses only a single set of comparators for memory access permissions checks that are switched in an iterative, time-multiplex style reducing hardware footprint by approx. 50% while maintaining full security features and RISC-V compatibility.

PMP Memory Accesses

Load/store accesses for which there are insufficient access permission do not trigger any memory/bus accesses at all. In contrast, instruction accesses for which there are insufficient access permission nevertheless lead to a memory/bus access (causing potential side effects on the memory side=. However, the fetched instruction will be discarded and the corresponding exception will still be triggered precisely.

3.6.32. `Sdext` ISA Extension

This ISA extension enables the RISC-V-compatible "external debug support" by implementing the CPU "debug mode", which is required for the on-chip debugger. See section On-Chip Debugger (OCD) / CPU Debug Mode for more information.

Table 70. Instructions and Timing

Class	Instructions	Execution cycles
System	`dret`	5

3.6.33. `Sdtrig` ISA Extension

This ISA extension implements the RISC-V-compatible "trigger module". See section On-Chip Debugger (OCD) / Trigger Module for more information.

3.7. Custom Functions Unit (CFU)

The Custom Functions Unit (CFU) is the central part of the NEORV32-specific `Zxcfu` ISA Extension and represents the actual hardware module that can be used to implement **custom RISC-V instructions**.

The CFU is intended for operations that are inefficient in terms of performance, latency, energy consumption or program memory requirements when implemented entirely in software. Some potential application fields and exemplary use-cases might include:

- **AI:** sub-word / vertical vector/SIMD operations like processing all four sub-bytes of a 32-bit data word individually
- **Cryptographic:** bit substitution and permutation
- **Communication:** data conversions like binary to gray-code
- **Arithmetic:** BCD (binary-coded decimal) operations; multiply-add operations; shift-and-add algorithms like CORDIC or BKM
- **Image processing:** look-up-tables for color space transformations
- implementing instructions from **other RISC-V ISA extensions** that are not yet supported by NEORV32

The NEORV32 CFU supports two different instruction formats (R3-type and R4-type; see CFU Instruction Formats) and also allows to implement up to 4 CFU-internal custom control and status registers (see CFU Control and Status Registers (CFU-CSRs)).

CFU Complexity

The CFU is not intended for complex and **CPU-independent** functional units that implement complete accelerators (like block-based AES encryption). These kind of accelerators should be implemented as memory-mapped co-processor via the Custom Functions Subsystem (CFS) to allow CPU-independent operation. A comparative survey of all NEORV32-specific hardware extension/customization options is provided in the user guide section Adding Custom Hardware Modules.

Default CFU Hardware Example

The default CFU module (`rtl/core/neorv32_cpu_cp_cfu.vhd`) implements the *Extended Tiny Encryption Algorithm (XTEA)* as "real world" application example.

3.7.1. CFU Instruction Formats

The custom instructions executed by the CFU utilize a specific opcode space in the `rv32` 32-bit instruction encoding space that has been explicitly reserved for user-defined extensions by the RISC-V specifications ("Guaranteed Non-Standard Encoding Space"). The NEORV32 CFU uses the `custom-0` and `custom-1` opcodes to identify the instruction implemented by the CFU and to differentiate between the predefined instruction formats.

The NEORV32 CFU utilizes these two opcodes to support user-defined **R3-type** instructions (2 source registers, 1 destination register) and **R4-type** instructions (3 source registers, 1 destination register). Both instruction formats are compliant to the RISC-V specification.

- `custom-0`: `0001011` RISC-V standard, used for NEORV32 CFU R3-Type Instructions (3x register addresses)
- `custom-1`: `0101011` RISC-V standard, used for NEORV32 CFU R4-Type Instructions (4x register addresses)

The provided instructions formats are *predefined* to allow an easy integration framework. However, system designers are free to ignore these and use their own instruction types and formats.

CFU R3-Type Instructions

The R3-type CFU instructions operate on two source registers `rs1` and `rs2` and return the processing result to the destination register `rd`. The actual operation can be defined by using the `funct7` and `funct3` bit fields. These immediates can also be used to pass additional data to the CFU like offsets, look-up-tables addresses or shift-amounts. However, the actual functionality is entirely user-defined. Note that all immediate values are always compile-time-static.

Example operation: `rd ⇐ rs1 xnor rs2` (bit-wise logical XNOR)

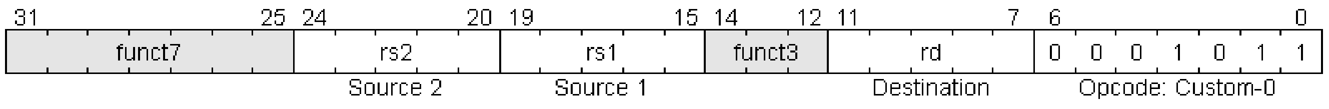


Figure 10. CFU R3-type instruction format

- `funct7`: 7-bit immediate (immediate data or function select)
- `rs2`: address of second source register (providing 32-bit source data)
- `rs1`: address of first source register (providing 32-bit source data)
- `funct3`: 3-bit immediate (immediate data or function select)
- `rd`: address of destination register (32-bit processing result)
- `opcode`: `0001011` (RISC-V `custom-0` opcode)

Instruction encoding space

By using the `funct7` and `funct3` bit fields entirely for selecting the actual operation a total of 1024 custom R3-type instructions can be implemented (7-bit + 3-bit = 10 bit → 1024 different values).

CFU R4-Type Instructions

The R4-type CFU instructions operate on three source registers `rs1`, `rs2` and `rs2` and return the processing result to the destination register `rd`. The actual operation can be defined by using the `funct3` bit field. Alternatively, this immediate can also be used to pass additional data to the CFU like offsets, look-up-tables addresses or shift-amounts. However, the actual functionality is entirely user-defined. Note that all immediate values are always compile-time-static.

Example operation: `rd <= (rs1 * rs2 + rs3)[31:0]` (multiply-and-accumulate; "MAC")

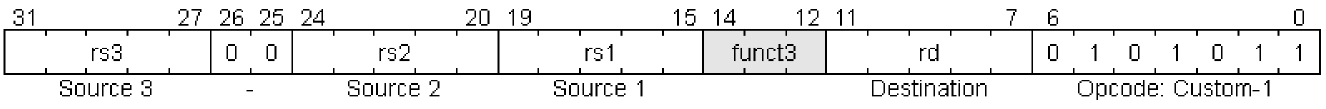


Figure 11. CFU R4-type instruction format

- `rs3`: address of third source register (providing 32-bit source data)
- `rs2`: address of second source register (providing 32-bit source data)
- `rs1`: address of first source register (providing 32-bit source data)
- `funct3`: 3-bit immediate (immediate data or function select)
- `rd`: address of destination register (32-bit processing result)
- `opcode`: `0101011` (RISC-V `custom-1` opcode)
- ▯▯ bits [26:25] of the R4-type instruction word are unused. However, these bits are ignored by CPU's instruction decoder and can be retrieved via the CFU's `funct7_i(6 downto 5)` signal.

Instruction encoding space

By using the `funct3` bit field entirely for selecting the actual operation a total of 8 custom R4-type instructions can be implemented (3-bit → 8 different values).

Re-purposing R4-type instructions as additional R3-type instructions

Advanced user can use the custom-1 opcode to implement additional R3-type instructions instead of the predefined r4-type instructions.

3.7.2. Using Custom Instructions in Software

The custom instructions provided by the CFU can be used in plain C code by using **intrinsics**. Intrinsics behave like "normal" C functions but under the hood they are a set of macros that hide the complexity of inline assembly, which is used to construct the custom 32-bit instruction words. Using intrinsics removes the need to modify the compiler, built-in libraries or the assembler when using custom instructions. Each intrinsic will be compiled into a single 32-bit instruction word without any overhead providing maximum code efficiency.

The NEORV32 software framework provides two pre-defined prototypes for custom instructions, which are defined in `sw/lib/include/neorv32_cpu_cfu.h`:

Listing 12. CFU instruction prototypes

```
uint32_t neorv32_cfu_r3_instr(funct7, funct3, rs1, rs2); // R3-type instructions
uint32_t neorv32_cfu_r4_instr(funct3, rs1, rs2, rs3);    // R4-type instructions
```

The intrinsic functions always return a 32-bit value of type `uint32_t` (the processing result), which can be discarded if not needed. Each intrinsic function requires several arguments depending on the instruction type/format:

- `funct7` - 7-bit immediate (R3-type)
- `funct3` - 3-bit immediate (R3-type, R4-type)
- `rs1` - source operand 1, 32-bit (R3-type, R4-type)
- `rs2` - source operand 2, 32-bit (R3-type, R4-type)
- `rs3` - source operand 3, 32-bit (R4-type)

The `funct3` and `funct7` bit-fields are used to pass 3-bit or 7-bit literals to the CFU. The `rs1`, `rs2` and `rs3` arguments pass the actual data to the CFU via register addresses. These register arguments can be populated with variables or literals; the compiler will add the required code to move the data into a register before passing it to the CFU. The following examples shows how to pass arguments:

Listing 13. CFU instruction usage example

```
uint32_t tmp = some_function();
...
uint32_t res = neorv32_cfu_r3_instr(0b0000000, 0b101, tmp, 123);
uint32_t foo = neorv32_cfu_r4_instr(0b011, tmp, res, (uint32_t)some_array[i]);
neorv32_cfu_r3_instr(0b0100100, 0b001, tmp, foo); // discard result
```

CFU Example Program

There is an example program for the CFU, which shows how to use the *default* CFU hardware module. This example program is located in `sw/example/demo_cfu`.

3.7.3. CFU Control and Status Registers (CFU-CSRs)

The CPU provides up to four control and status registers (`cfureg*`) to be used within the CFU. These CSRs are mapped to the "custom user-mode read/write" CSR address space, which is explicitly reserved for platform-specific application by the RISC-V spec. For example, these CSRs can be used to pass additional operands to the CFU, to obtain additional results, to check processing status or to configure operation modes.

Listing 14. CFU CSR Access Example

```
neorv32_cpu_csr_write(CSR_CFUREG0, 0xabcdabcd); // write data to CFU CSR 0
uint32_t tmp = neorv32_cpu_csr_read(CSR_CFUREG3); // read data from CFU CSR 3
```

Additional CFU-internal CSRs

If more than four CFU-internal CSRs are required the designer can implement an "indirect access mechanism" based on just two of the default CSRs: one CSR is used to configure the index while the other is used as alias to exchange data with the indexed CFU-internal CSR - this concept is similar to the RISC-V Indirect CSR Access Extension Specification (`Smcsrind`).

Security Considerations

The CFU CSRs are mapped to the user-mode CSR space so software running at *any privilege level* can access these CSRs.

3.7.4. Custom Instructions Hardware

The actual functionality of the CFU's custom instructions is defined by the user-defined logic inside the CFU hardware module (`rtl/core/neorv32_cpu_cp_cfu.vhd`). This file is highly commented to explain the interface and to illustrate hardware design considerations.

CFU operations can be entirely combinatorial (like bit-reversal) so the result is available at the end of the current clock cycle. However, operations can also take several clock cycles to complete (like multiplications) and may also include internal states and memories.

CFU Hardware Resource Requirements

Enabling the CFU and actually implementing R4-type instructions (or more precisely, using the third register source `rs3`) will add an additional read port to the core's register file increasing resource requirements of the register file by 50%.

CFU Execution Time

The CFU has to complete computation within a **bound time window** (default = 512 clock cycles). Otherwise, the CFU operation is terminated by the CPU execution logic and an illegal instruction exception is raised. See section CPU Arithmetic Logic Unit for more information.

CFU Exception

The CFU can intentionally raise an illegal instruction exception by not asserting the `done` at all causing an execution timeout. For example this can be used to signal invalid configurations/operations to the runtime environment. See the documentation in the CFU's VHDL source file for more information.

3.8. Control and Status Registers (CSRs)

The following table shows a summary of all available NEORV32 CSRs. The address field defines the CSR address for the CSR access instructions. The "Name [ASM]" column provides the CSR name aliases that can be used in (inline) assembly. The "Name [C]" column lists the name aliases that are defined by the NEORV32 core library. These can be used in plain C code. The "Access" column shows the minimal required privilege mode required for accessing the according CSR (`M` = machine-mode, `U` = user-mode, `D` = debug-mode) and the read/write capabilities (`RW` = read-write, `RO` = read-only)

Unused, Reserved, Unimplemented and Disabled CSRs

All CSRs and CSR bits that are not listed in the table below are *unimplemented* and are *hardwired to zero*. Additionally, CSRs that are unavailable ("disabled") because the according ISA extension is not enabled are also considered *unimplemented* and are also hardwired to zero. Any access to such a CSR will raise an illegal instruction exception. All writable CSRs provide **WARL** behavior (write all values; read only legal values). Application software should always read back a CSR after writing to check if the targeted bits can actually be modified.

Table 71. NEORV32 Control and Status Registers (CSRs)

Address	Name [ASM]	Name [C]	Access	Description
Floating-Point CSRs				
0x001	`fflags`	`CSR_FFLAGS`	URW	Floating-point accrued exceptions
0x002	`frm`	`CSR_FRM`	URW	Floating-point dynamic rounding mode
0x003	`fcsr`	`CSR_FCSR`	URW	Floating-point control and status
Machine Trap Setup CSRs				
0x300	`mstatus`	`CSR_MSTATUS`	MRW	Machine status register - low word
0x301	`misa`	`CSR_MISA`	MRW	Machine CPU ISA and extensions
0x304	`mie`	`CSR_MIE`	MRW	Machine interrupt enable register

Copyright by Stephan Nolting. All rights reserved.

Address	Name [ASM]	Name [C]	Access	Description
0x305	`mtvec`	`CSR_MTVEC`	MRW	Machine trap-handler base address for ALL traps
0x306	`mcounteren`	`CSR_MCOUNTEREN`	MRW	Machine counter-enable register
0x310	`mstatush`	`CSR_MSTATUSH`	MRW	Machine status register - high word
Machine Configuration CSRs				
0x30a	`menvcfg`	`CSR_MENVCFG`	MRW	Machine environment configuration register - low word
0x31a	`menvcfgh`	`CSR_MENVCFGH`	MRW	Machine environment configuration register - high word
Machine Counter Setup CSRs				
0x320	`mcountinhibit`	`CSR_MCOUNTINHIBIT`	MRW	Machine counter-inhibit register
Machine Trap Handling CSRs				
0x340	`mscratch`	`CSR_MSCRATCH`	MRW	Machine scratch register
0x341	`mepc`	`CSR_MEPC`	MRW	Machine exception program counter
0x342	`mcause`	`CSR_MCAUSE`	MRW	Machine trap cause
0x343	`mtval`	`CSR_MTVAL`	MRW	Machine trap value
0x344	`mip`	`CSR_MIP`	MRW	Machine interrupt pending register

Address	Name [ASM]	Name [C]	Access	Description
0x34a	`mtinst`	`CSR_MTINST`	MRW	Machine trap instruction
		Machine Physical Memory Protection CSRs		
0x3a0 .. 0x303	`pmpcfg0` .. `pmpcfg3`	`CSR_PMPCFG0` .. `CSR_PMPCFG3`	MRW	Physical memory protection configuration registers
0x3b0 .. 0x3bf	`pmpaddr0` .. `pmpaddr15`	`CSR_PMPADDR0` .. `CSR_PMPADDR15`	MRW	Physical memory protection address registers
		Trigger Module CSRs		
0x7a0	`tselect`	`CSR_TSELECT`	MRW	Trigger select register
0x7a1	`tdata1`	`CSR_TDATA1`	MRW	Trigger data register 1
0x7a2	`tdata2`	`CSR_TDATA2`	MRW	Trigger data register 2
0x7a4	`tinfo`	`CSR_TINFO`	MRW	Trigger information register
		CPU Debug Mode CSRs		
0x7b0	`dcsr`	-	DRW	Debug control and status register
0x7b1	`dpc`	-	DRW	Debug program counter
0x7b2	`dscratch0`	-	DRW	Debug scratch register 0
		(Machine) Counter and Timer CSRs		
0xb00	`mcycle`	`CSR_MCYCLE`	MRW	Machine cycle counter low word

Address	Name [ASM]	Name [C]	Access	Description
0xb02	`minstret`	`CSR_MINSTRET`	MRW	Machine instruction-retired counter low word
0xb80	`mcycleh`	`CSR_MCYCLEH`	MRW	Machine cycle counter high word
0xb82	`minstreth`	`CSR_MINSTRETH`	MRW	Machine instruction-retired counter high word
0xc00	`cycle`	`CSR_CYCLE`	URO	Cycle counter low word
0xc02	`instret`	`CSR_INSTRET`	URO	Instruction-retired counter low word
0xc80	`cycleh`	`CSR_CYCLEH`	URO	Cycle counter high word
0xc82	`instreth`	`CSR_INSTRETH`	URO	Instruction-retired counter high word
Hardware Performance Monitors (HPM) CSRs				
0x323 .. 0x32f	`mhpmevent3` .. `mhpmevent15`	`CSR_MHPMEVENT3` .. `CSR_MHPMEVENT15`	MRW	Machine performance-monitoring event select for counter 3..15
0xb03 .. 0xb0f	`mhpmcounter3` .. `mhpmcounter15`	`CSR_MHPMCOUNTER3` .. `CSR_MHPMCOUNTER15`	MRW	Machine performance-monitoring counter 3..15 low word
0xb83 .. 0xb8f	`mhpmcounter3h` .. `mhpmcounter15h`	`CSR_MHPMCOUNTER3H` .. `CSR_MHPMCOUNTER15H`	MRW	Machine performance-monitoring counter 3..15 high word
Machine Information CSRs				
0xf11	**`mvendorid`**	`CSR_MVENDORID`	MRO	Machine vendor ID
0xf12	**`marchid`**	`CSR_MARCHID`	MRO	Machine architecture ID
0xf13	**`mimpid`**	`CSR_MIMPID`	MRO	Machine implementation ID / version
0xf14	**`mhartid`**	`CSR_MHARTID`	MRO	Machine hardware thread ID

Address	Name [ASM]	Name [C]	Access	Description
0xf15	`mconfigptr`	`CSR_MCONFIGPTR`	MRO	Machine configuration pointer register
NEORV32-Specific CSRs				
0xbc0	`mxiccsreg`	`CSR_MXICCSREG`	MRW	Inter-core communication status register
0xbc1	`mxiccdata`	`CSR_MXICCDATA`	MRW	Inter-core communication data register
0x800 .. 0x803	`cfureg0` .. `cfureg3`	`CSR_CFUCREG0` .. `CSR_CFUCREG3`	URW	Custom CFU registers 0 to 3
0xfc0	`mxisa`	`CSR_MXISA`	MRO	Extended machine CPU ISA and extensions

Copyright by Stephan Nolting. All rights reserved.

3.8.1. Floating-Point CSRs

`fflags`

Name	Floating-point accrued exceptions
Address	`0x001`
Reset value	`0x00000000`
ISA	`Zicsr` & `Zfinx`
Description	FPU status flags.

Table 72. `fflags` CSR bits

Bit	R/W	Function
0	r/w	**NX**: inexact
1	r/w	**UF**: underflow
2	r/w	**OF**: overflow
3	r/w	**DZ**: division by zero
4	r/w	**NV**: invalid operation

`frm`

Name	Floating-point dynamic rounding mode
Address	`0x002`
Reset value	`0x00000000`
ISA	`Zicsr` & `Zfinx`
Description	The `frm` CSR is used to configure the rounding mode of the FPU.

Table 73. `frm` CSR bits

Bit	R/W	Function
2:0	r/w	Rounding mode

`fcsr`

Name	Floating-point control and status register

Address	`0x003`
Reset value	`0x00000000`
ISA	`Zicsr` & `Zfinx`
Description	The `fcsr` provides combined access to the **`fflags`** and **`frm`** flags.

Table 74. `fcsr` CSR bits

Bit	R/W	Function
4:0	r/w	Accrued exception flags (**`fflags`**)
7:5	r/w	Rounding mode (**`frm`**)

3.8.2. Machine Trap Setup CSRs

mstatus

Name	Machine status register - low word
Address	`0x300`
Reset value	`0x00001800`
ISA	`Zicsr`
Description	The `mstatus` CSR is used to configure general machine environment parameters.

Table 75. `mstatus` CSR bits

Bit	Name [C]	R/W	Function
3	`CSR_MSTATUS_MIE`	r/w	**MIE**: Machine-mode interrupt enable flag
7	`CSR_MSTATUS_MPIE`	r/w	**MPIE**: Previous machine-mode interrupt enable flag state
12:11	`CSR_MSTATUS_MPP_H` : `CSR_MSTATUS_MPP_L`	r/w	**MPP**: Previous machine privilege mode, `11` = machine-mode "M", `00` = user-mode "U"; other values will fall-back to machine-mode
17	`CSR_MSTATUS_MPRV`	r/w	**MPRV**: Effective privilege mode for load/stores; use `MPP` as effective privilege mode when set; hardwired to zero if user-mode not implemented
21	`CSR_MSTATUS_TW`	r/w	**TW**: Trap on execution of `wfi` instruction in user mode when set; hardwired to zero if user-mode not implemented

If the core is in user-mode, machine-mode interrupts are globally **enabled** even if `mstatus.mie` is cleared: "Interrupts for higher-privilege modes, y>x, are always globally enabled regardless of the setting of the global yIE bit for the higher-privilege mode." - RISC-V ISA Spec.

misa

Name	ISA and extensions
Address	`0x301`
Reset value	`DEFINED`, according to enabled ISA extensions
ISA	`Zicsr`
Description	The `misa` CSR provides information regarding the availability of basic RISC-V ISa extensions.

The NEORV32 `misa` CSR is read-only. Hence, active CPU extensions are entirely defined by pre-synthesis configurations and cannot be switched on/off during runtime. For compatibility reasons any write access to this CSR is simply ignored and will *not* cause an illegal instruction exception.

Table 76. `misa` CSR bits

Bit	Name [C]	R/W	Function
1	`CSR_MISA_B_EXT`	r/-	**B**: CPU extension (bit-manipulation) available, set when `B` ISA Extension enabled
2	`CSR_MISA_C_EXT`	r/-	**C**: CPU extension (compressed instruction) available, set when `C` ISA Extension enabled
4	`CSR_MISA_E_EXT`	r/-	**E**: CPU extension (embedded) available, set when `E` ISA Extension enabled
8	`CSR_MISA_I_EXT`	r/-	**I**: CPU base ISA, cleared when `E` ISA Extension enabled
12	`CSR_MISA_M_EXT`	r/-	**M**: CPU extension (mul/div) available, set when `M` ISA Extension enabled
20	`CSR_MISA_U_EXT`	r/-	**U**: CPU extension (user mode) available, set when `U` ISA Extension enabled
23	`CSR_MISA_X_EXT`	r/-	**X**: bit is always set to indicate non-standard / NEORV32-specific extensions
31:30	`CSR_MISA_MXL_HI_EXT` : `CSR_MISA_MXL_LO_EXT`	r/-	**MXL**: 32-bit architecture indicator (always `01`)

Machine-mode software can discover available `Z*` *sub-extensions* (like `Zicsr` or `Zfinx`) by checking the NEORV32-specific **`mxisa`** CSR.

`mie`

Name	Machine interrupt-enable register
Address	`0x304`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mie` CSR is used to enable/disable individual interrupt sources.

Table 77. `mie` CSR bits

Bit	Name [C]	R/W	Function
3	`CSR_MIE_MSIE`	r/w	**MSIE**: Machine *software* interrupt enable (from Core Local Interruptor (CLINT))
7	`CSR_MIE_MTIE`	r/w	**MTIE**: Machine *timer* interrupt enable (from Core Local Interruptor (CLINT))
11	`CSR_MIE_MEIE`	r/w	**MEIE**: Machine *external* interrupt enable
31:16	`CSR_MIE_FIRQ15E` : `CSR_MIE_FIRQ0E`	r/w	Fast interrupt channel 15..0 enable

`mtvec`

Name	Machine trap-handler base address
Address	`0x305`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mtvec` CSR holds the trap vector configuration.

Table 78. `mtvec` CSR bits

Bit	R/W	Function
1:0	r/w	**MODE**: mode configuration, `00` = DIRECT, `01` = VECTORED; other encodings are *reserved*.
31:2	r/w	**BASE**: in DIRECT mode = 4-byte-aligned base address of trap base handler, *all* traps jump to `pc = BASE`; in VECTORED mode = 128-byte-aligned base address of trap vector table, interrupts cause a jump to `pc = BASE + 4 * mcause` and exceptions a jump to `pc = BASE`.

Interrupt Latency

The vectored `mtvec` mode is useful for reducing the time between interrupt request (IRQ) and servicing it (ISR). As software does not need to determine the interrupt cause the reduction in latency can be 5 to 10 times and as low as *26* cycles.

`mcounteren`

Name	Machine counter enable
Address	`0x306`

Reset value	`0x00000000`
ISA	`Zicsr` & `Zicntr` & `U`
Description	The `mcounteren` CSR is used to constrain user-mode access to the CPU's counter CSRs.

Table 79. `mcounteren` CSR bits

Bit	Name [C]	R/W
Function	0	`CSR_MCOUNTEREN_CY`
r/w	**CY**: User-mode is allowed to read **`cycle[h]`** CSRs when set	1
-	r/-	**TM**: not implemented, hardwired to zero
2	`CSR_MCOUNTEREN_IR`	r/w
IR: User-mode is allowed to read **`instret[h]`** CSRs when set	31:3	-

`mstatush`

Name	Machine status register - high word
Address	`0x310`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The features of this CSR are not implemented yet. The register is read-only and always returns zero.

3.8.3. Machine Trap Handling CSRs

mscratch

Name	Scratch register for machine trap handlers
Address	`0x340`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mscratch` is a general-purpose machine-mode scratch register.

mepc

Name	Machine exception program counter
Address	`0x341`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mepc` CSR provides the instruction address where execution has stopped/failed when an interrupt is triggered / an exception is raised. See section Traps, Exceptions and Interrupts for a list of all legal values. The `mret` instruction will return to the address stored in `mepc` by automatically moving `mepc` to the program counter.

`mepc[0]` is hardwired to zero. If IALIGN = 32 (i.e. `C` ISA Extension is disabled) then `mepc[1]` is also hardwired to zero.

mcause

Name	Machine trap cause
Address	`0x342`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mcause` CSRs shows the exact cause of a trap. See section Traps, Exceptions and Interrupts for a list of all legal values.

Table 80. `mcause` CSR bits

Bit	R/W	Function
4:0	r/w	**Exception code**: see NEORV32 Trap Listing
31	r/w	**Interrupt**: `1` if the trap is caused by an interrupt (`0` if the trap is caused by an exception)

`mtval`

Name	Machine trap value
Address	`0x343`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mtval` CSR provides additional information why a trap was entered. See section Traps, Exceptions and Interrupts for more information.

Read-Only

Note that the NEORV32 `mtval` CSR is updated by the hardware only and cannot be written from software. However, any write-access will be ignored and will not cause an exception to maintain RISC-V compatibility.

`mip`

Name	Machine interrupt pending
Address	`0x344`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mip` CSR shows currently *pending* machine-mode interrupt requests. Any write access to this register is ignored.

Table 81. `mip` CSR bits

Bit	Name [C]	R/W	Function
3	`CSR_MIP_MSIP`	r/-	**MSIP**: Machine *software* interrupt pending, triggered by `msi_i` top port (see CPU Top Entity - Signals); cleared by source-specific mechanism

Bit	Name [C]	R/W	Function
7	`CSR_MIP_MTIP`	r/-	**MTIP**: Machine *timer* interrupt pending, triggered by `mei_i` top port (see CPU Top Entity - Signals) or by the processor-internal Core Local Interruptor (CLINT); cleared by source-specific mechanism
11	`CSR_MIP_MEIP`	r/-	**MEIP**: Machine *external* interrupt pending, triggered by `mti_i` top port (see CPU Top Entity - Signals) or by the processor-internal Core Local Interruptor (CLINT); cleared by source-specific mechanism
31:16	`CSR_MIP_FIRQ15P` : `CSR_MIP_FIRQ0P`	r/-	**FIRQxP**: Fast interrupt channel 15..0 pending, see NEORV32-Specific Fast Interrupt Requests; cleared by source-specific mechanism

FIRQ Channel Mapping

See section NEORV32-Specific Fast Interrupt Requests for the mapping of the FIRQ channels and the according interrupt-triggering processor module.

`mtinst`

Name	Machine trap instruction
Address	`0x34a`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The `mtinst` CSR provides additional information why a trap was entered. See section Traps, Exceptions and Interrupts for more information.

Read-Only

Note that the NEORV32 `mtinst` CSR is updated by the hardware only and cannot be written from software. However, any write-access will be ignored and will not cause an exception to maintain RISC-V compatibility.

Instruction Transformation

The RISC-V priv. spec. suggests that the instruction word written to `mtinst` by the hardware should be "transformed". However, the NEORV32 `mtinst` CSR uses a simplified transformation scheme: if the trap-causing instruction is a standard 32-bit instruction, `mtinst` contains the exact instruction word that caused the trap. If the trap-causing instruction is a compressed instruction, `mtinst` contains the de-compressed 32-bit equivalent with bit 1 being cleared while all remaining bits represent the pre-decoded 32-bit instruction equivalent.

3.8.4. Machine Configuration CSRs

`menvcfg`

Name	Machine environment configuration register - low word
Address	`0x30a`
Reset value	`0x00000000`
ISA	`Zicsr` & `U`
Description	Currently, the features of this CSR are not supported. Hence, the entire register is hardwired to all-zero.

`menvcfgh`

Name	Machine environment configuration register - high word
Address	`0x31a`
Reset value	`0x00000000`
ISA	`Zicsr` & `U`
Description	Currently, the features of this CSR are not supported. Hence, the entire register is hardwired to all-zero.

3.8.5. Machine Physical Memory Protection CSRs

The physical memory protection system is configured via the `PMP_NUM_REGIONS` and `PMP_MIN_GRANULARITY` top entity generics. `PMP_NUM_REGIONS` defines the total number of implemented regions. Note that the maximum number of regions is constrained to 16. If trying to access a PMP-related CSR beyond `PMP_NUM_REGIONS` **no illegal instruction exception** is triggered. The according CSRs are read-only (writes are ignored) and always return zero. See section `Smpmp` ISA Extension for more information.

`pmpcfg`

Name	Physical memory protection region configuration registers
Address	`0x3a0` (`pmpcfg0`) `0x3a1` (`pmpcfg1`) `0x3a2` (`pmpcfg2`) `0x3a3` (`pmpcfg3`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Smpmp`
Description	Configuration of physical memory protection regions. Each region provides an individual 8-bit array in these CSRs.

Table 82. `pmpcfg*` CSR Bits

Bit	Name [C]	R/W	Function
0	`PMPCFG_R`	r/w	**R**: Read permission
1	`PMPCFG_W`	r/w	**W**: Write permission
2	`PMPCFG_X`	r/w	**X**: Execute permission
4:3	`PMPCFG_A_MSB` : `PMPCFG_A_LSB`	r/w	**A**: Mode configuration (`00` = OFF, `01` = TOR, `10` = NA4, `11` = NAPOT)
7	`PMPCFG_L`	r/w	**L**: Lock bit, prevents further write accesses, also enforces access rights in machine-mode, can only be cleared by CPU reset

Implemented Modes

In order to reduce the CPU size certain PMP modes (`A` bits) can be excluded from synthesis. Use the `PMP_TOR_MODE_EN` and `PMP_NAP_MODE_EN` Processor Top Entity - Generics to control implementation of the according modes.

`pmpaddr`

Name	Physical memory protection region address registers
Address	`0x3b0` (`pmpaddr1`) `0x3b1` (`pmpaddr2`) `0x3b2` (`pmpaddr3`) `0x3b3` (`pmpaddr4`) `0x3b4` (`pmpaddr5`) `0x3b5` (`pmpaddr6`) `0x3b6` (`pmpaddr6`) `0x3b7` (`pmpaddr7`) `0x3b8` (`pmpaddr8`) `0x3b9` (`pmpaddr9`) `0x3ba` (`pmpaddr10`) `0x3bb` (`pmpaddr11`) `0x3bc` (`pmpaddr12`) `0x3bd` (`pmpaddr13`) `0x3be` (`pmpaddr14`) `0x3bf` (`pmpaddr15`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Smpmp`
Description	Region address/boundaries configuration.

Table 83. `pmpaddr*` CSR Bits

Bit	**R/W**	**Description**	**31:30**
r-w	address bits `33 downto 32´`, hardwired to zero	29:0	r/w

3.8.6. (Machine) Counter and Timer CSRs

`time[h]` CSRs (Wall Clock Time)

The NEORV32 does not implement the user-mode `time[h]` registers. Any access to these registers will trap. It is recommended that the trap handler software provides a means of accessing the machine timer oft the Core Local Interruptor (CLINT).

Instruction Retired Counter Increment

The `[m]instret[h]` counter always increments when a instruction enters the pipeline's execute stage no matter if this instruction is actually going to retire or if it causes an exception.

`cycle[h]`

Name	Cycle counter
Address	`0xc00` (`cycle`) `0xc80` (`cycleh`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Zicntr`
Description	The `cycle[h]` CSRs are user-mode shadow copies of the according **`mcycle[h]`** CSRs. The user-mode counter are read-only. Any write access will raise an illegal instruction exception.

`instret[h]`

Name	Instructions-retired counter
Address	`0xc02` (`instret`) `0xc82` (`instreth`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Zicntr`
Description	The `instret[h]` CSRs are user-mode shadow copies of the according **`minstret[h]`** CSRs. The user-mode counter are read-only. Any write access will raise an illegal instruction exception.

`mcycle[h]`

Name	Machine cycle counter
Address	`0xb00` (`mcycle`)
	`0xb80` (`mcycleh`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Zicntr`
Description	If not halted via the **`mcountinhibit`** CSR the `cycle[h]` CSRs will increment with every active CPU clock cycle (CPU not in sleep mode). These registers are read/write only for machine-mode software.

`minstret[h]`

Name	Machine instructions-retired counter
Address	`0xb02` (`minstret`)
	`0xb82` (`minstreth`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Zicntr`
Description	If not halted via the **`mcountinhibit`** CSR the `minstret[h]` CSRs will increment with every retired instruction. These registers are read/write only for machine-mode software

Instruction Retiring

Note that **all** executed instruction do increment the `[m]instret`[h] counters even if they do not retire (e.g. if the instruction causes an exception).

3.8.7. Hardware Performance Monitors (HPM) CSRs

Machine-Mode HPMs Only

Note that only the machine-mode hardware performance counter CSR are available (mhpmcounter*[h]). Accessing any user-mode HPM CSR (hpmcounter*[h]) will raise an illegal instruction exception.

The actual number of implemented hardware performance monitors is configured via the HPM_NUM_CNTS top entity generic, Note that always all 13 HPM counter and configuration registers (mhpmcounter*[h]) are implemented, but only the actually configured ones are implemented as "real" physical registers - the remaining ones will be hardwired to zero. If trying to access an HPM-related CSR beyond HPM_NUM_CNTS **no illegal instruction exception is triggered**. These CSRs are read-only, writes are ignored and reads always return zero.

The total counter width of the HPMs can be configured before synthesis via the HPM_CNT_WIDTH generic (0..64-bit). If HPM_NUM_CNTS is less than 64, all remaining MSB-aligned bits are hardwired to zero.

mhpmevent

Name	Machine hardware performance monitor event select
Address	0x233 (mhpmevent3)
	0x234 (mhpmevent4)
	0x235 (mhpmevent5)
	0x236 (mhpmevent6)
	0x237 (mhpmevent7)
	0x238 (mhpmevent8)
	0x239 (mhpmevent9)
	0x23a (mhpmevent10)
	0x23b (mhpmevent11)
	0x23c (mhpmevent12)
	0x23d (mhpmevent13)
	0x23e (mhpmevent14)
	0x23f (mhpmevent15)
Reset value	0x00000000
ISA	Zicsr & Zihpm

Description	The value in these CSRs define the architectural events that cause an increment of the according `mhpmcounter*[h]` counter(s). All available events are listed in the table below. If more than one event is selected, the according counter will increment if *any* of the enabled events is observed (logical OR). Note that the counter will only increment by 1 step per clock cycle even if more than one trigger event is observed.

Table 84. `mhpmevent*` CSR Bits

Bit	Name [C]	R/W	Event Description
			RISC-V-compatible
0	`HPMCNT_EVENT_CY`	r/w	active clock cycle (CPU not in Sleep Mode)
1	`HPMCNT_EVENT_TM`	r/-	*not implemented*, hardwired to zero
2	`HPMCNT_EVENT_IR`	r/w	any executed instruction (16-bit/compressed or 32-bit/uncompressed)
			NEORV32-specific
3	`HPMCNT_EVENT_COMPR`	r/w	any executed 16-bit/compressed (`C` ISA Extension) instruction
4	`HPMCNT_EVENT_WAIT_DIS`	r/w	instruction dispatch wait cycle (wait for instruction prefetch-buffer refill (CPU Control Unit IPB); caused by a fence instruction, a control flow transfer or a instruction fetch bus wait cycle)
5	`HPMCNT_EVENT_WAIT_ALU`	r/w	any delay/wait cycle caused by a *multi-cycle* CPU Arithmetic Logic Unit operation
6	`HPMCNT_EVENT_BRANCH`	r/w	any executed branch instruction (unconditional, conditional-taken or conditional-not-taken)
7	`HPMCNT_EVENT_BRANCHED`	r/w	any control transfer operation (unconditional jump, taken conditional branch or trap entry/exit)
8	`HPMCNT_EVENT_LOAD`	r/w	any executed load operation (including any atomic memory operations)
9	`HPMCNT_EVENT_STORE`	r/w	any executed store operation (including any atomic memory operations)
10	`HPMCNT_EVENT_WAIT_LSU`	r/w	any memory/bus/cache/etc. delay/wait cycle while executing any load or store operation (caused by a data bus wait cycle))
11	`HPMCNT_EVENT_TRAP`	r/w	starting processing of any trap (Traps, Exceptions and Interrupts)

Instruction Retiring ("Retired == Executed")

The CPU HPM/counter logic treats all executed instruction as "retired" even if they raise an exception, cause an interrupt, trigger a privilege mode change or were not meant to retire (i.e. claimed by the RISC-V spec.).

mhpmcounter[h]

Name	Machine hardware performance monitor (HPM) counter
Address	`0xb03`, `0xb83` (`mhpmcounter3`, `mhpmcounter3h`)
	`0xb04`, `0xb84` (`mhpmcounter4`, `mhpmcounter4h`)
	`0xb05`, `0xb85` (`mhpmcounter5`, `mhpmcounter5h`)
	`0xb06`, `0xb86` (`mhpmcounter6`, `mhpmcounter6h`)
	`0xb07`, `0xb87` (`mhpmcounter7`, `mhpmcounter7h`)
	`0xb08`, `0xb88` (`mhpmcounter8`, `mhpmcounter8h`)
	`0xb09`, `0xb89` (`mhpmcounter9`, `mhpmcounter9h`)
	`0xb0a`, `0xb8a` (`mhpmcounter10`, `mhpmcounter10h`)
	`0xb0b`, `0xb8b` (`mhpmcounter11`, `mhpmcounter11h`)
	`0xb0c`, `0xb8c` (`mhpmcounter12`, `mhpmcounter12h`)
	`0xb0d`, `0xb8d` (`mhpmcounter13`, `mhpmcounter13h`)
	`0xb0e`, `0xb8e` (`mhpmcounter14`, `mhpmcounter14h`)
	`0xb0f`, `0xb8f` (`mhpmcounter15`, `mhpmcounter15h`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Zihpm`
Description	If not halted via the **`mcountinhibit`** CSR the HPM counter CSR(s) increment whenever the configured event from the according **`mhpmevent`** CSR occurs. The counter registers are read/write for machine mode and are not accessible for lower-privileged software.

3.8.8. Machine Counter Setup CSRs

`mcountinhibit`

Name	Machine counter-inhibit register
Address	`0x320`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	Set bit to halt the according counter CSR.

Table 85. `mcountinhibit` CSR Bits

Bit	Name [C]	R/W	Description
0	`CSR_MCOUNTINHIBIT_IR`	r/w	**IR**: Set to `1` to halt `[m]instret[h]`; hardwired to zero if `Zicntr` ISA extension is disabled
1	-	r/-	**TM**: Hardwired to zero as `time[h]` CSRs are not implemented
2	`CSR_MCOUNTINHIBIT_CY`	r/w	**CY**: Set to `1` to halt `[m]cycle[h]`; hardwired to zero if `Zicntr` ISA extension is disabled
15:3	`CSR_MCOUNTINHIBIT_HPM3` : `CSR_MCOUNTINHIBIT_HPM15`	r/w	**HPMx**: Set to `1` to halt `[m]hpmcount*[h]`; hardwired to zero if `Zihpm` ISA extension is disabled

3.8.9. Machine Information CSRs

mvendorid

Name	Machine vendor ID
Address	`0xf11`
Reset value	`DEFINED`
ISA	`Zicsr`
Description	Vendor ID (JEDEC identifier, lowest 11 bits), assigned via the `JEDEC_ID` top generic (Processor Top Entity - Generics).

marchid

Name	Machine architecture ID
Address	`0xf12`
Reset value	`0x00000013`
ISA	`Zicsr`
Description	The `marchid` CSR is read-only and provides the NEORV32 official RISC-V open-source architecture ID (decimal: 19, 32-bit hexadecimal: 0x00000013).

mimpid

Name	Machine implementation ID
Address	`0xf13`
Reset value	`DEFINED`
ISA	`Zicsr`
Description	The `mimpid` CSR is read-only and provides the version of the NEORV32 as BCD-coded number (example: `mimpid = 0x01020312` → 01.02.03.12 → version 1.2.3.12).

mhartid

Name	Machine hardware thread ID
Address	`0xf14`

Reset value	`DEFINED`
ISA	`Zicsr`
Description	The `mhartid` CSR is read-only and provides the core's hart ID. For a multi-core system each core's hart ID is unique starting at 0 for the first core.

`mconfigptr`

Name	Machine configuration pointer register
Address	`0xf15`
Reset value	`0x00000000`
ISA	`Zicsr`
Description	The features of this CSR are not implemented yet. The register is read-only and always returns zero.

3.8.10. NEORV32-Specific CSRs

RISC-V-Compliant Mapping

All NEORV32-specific CSRs are mapped to addresses that are explicitly reserved for custom/implementation-specific use (assured by the RISC-V privileged specifications).

`cfureg`

Name	Custom (user-defined) CFU CSRs
Address	`0x800` (`cfureg0`) `0x801` (`cfureg1`) `0x802` (`cfureg2`) `0x803` (`cfureg3`)
Reset value	`0x00000000`
ISA	`Zicsr` & `Zxcfu`
Description	User-defined CSRs to be used within the Custom Functions Unit (CFU).

`mxiccsreg`

Name	Inter-Core Communication (ICC) status register
Address	`0xbc0`
Reset value	`0x40000000`
ISA	`Zicsr` & `X`
Description	Shows the status of the core's inter-core communication link (message queue / FIFO status flags). The entire CSR is read-only. However, write accesses are ignored. This CSR is hardwired to all-zero if the Dual-Core Configuration is disabled.

Table 86. `mxiccsreg` CSR Bits

Bit	Name [C]	R/W	Description
0	`CSR_MXICCSREG_RX_AVAIL`	r/-	Set if RX data from the other core is available.
1	`CSR_MXICCSREG_TX_FREE`	r/-	Set if there is free space for TX data for the other core.
31:2	-	r/-	Reserved; hardwired to zero.

`mxiccdata`

Name	Inter-Core Communication (ICC) data register
Address	`0xbc1`
Reset value	`0x00000000`
ISA	`Zicsr` & `X`
Description	This CSR provides access to the inter-core communication message queues that are implemented as simple FIFOs. Writing to this register will put data into the message queue so it can be read by the other core. Reading from this register will return data received from the other core (i.e. this CSR has side effects when reading). A read access will return all-zero of no RX data is available from the other core. This CSR is hardwired to all-zero if the Dual-Core Configuration is disabled.

`mxisa`

Name	Machine extended ISA and extensions register
Address	`0xfc0`
Reset value	`DEFINED`
ISA	`Zicsr` & `X`
Description	The `mxisa` CSRs is a NEORV32-specific read-only CSR that helps machine-mode software to discover additional ISA (sub-)extensions and CPU configuration options.

Table 87. `mxisa` CSR Bits

Bit	Name [C]	R/W	Description
0	`CSR_MXISA_ZICSR`	r/-	`Zicsr` ISA Extension available
1	`CSR_MXISA_ZIFENCEI`	r/-	`Zifencei` ISA Extension available
2	`CSR_MXISA_ZMMUL`	r/-	`Zmmul` - ISA Extension available
3	`CSR_MXISA_ZXCFU`	r/-	`Zxcfu` ISA Extension available
4	`CSR_MXISA_ZKT`	r/-	`Zkt` ISA Extension available
5	`CSR_MXISA_ZFINX`	r/-	`Zfinx` ISA Extension available
6	`CSR_MXISA_ZICOND`	r/-	`Zicond` ISA Extension available
7	`CSR_MXISA_ZICNTR`	r/-	`Zicntr` ISA Extension available
8	`CSR_MXISA_SMPMP`	r/-	`Smpmp` ISA Extension available

Bit	Name [C]	R/W	Description
9	`CSR_MXISA_ZIHPM`	r/-	`Zihpm` ISA Extension available
10	`CSR_MXISA_SDEXT`	r/-	`Sdext` ISA Extension available
11	`CSR_MXISA_SDTRIG`	r/-	`Sdtrig` ISA Extension available
12	`CSR_MXISA_ZBKX`	r/-	`Zbkx` ISA Extension available
13	`CSR_MXISA_ZKND`	r/-	`Zknd` ISA Extension available
14	`CSR_MXISA_ZKNE`	r/-	`Zkne` ISA Extension available
15	`CSR_MXISA_ZKNH`	r/-	`Zknh` ISA Extension available
16	`CSR_MXISA_ZBKB`	r/-	`Zbkb` ISA Extension available
17	`CSR_MXISA_ZBKC`	r/-	`Zbkc` ISA Extension available
18	`CSR_MXISA_ZKN`	r/-	`Zkn` ISA Extension available
19	`CSR_MXISA_ZKSH`	r/-	`Zksh` ISA Extension available
20	`CSR_MXISA_ZKSED`	r/-	`Zksed` ISA Extension available
21	`CSR_MXISA_ZKS`	r/-	`Zks` ISA Extension available
22	`CSR_MXISA_ZBA`	r/-	`Zba` ISA Extension available
23	`CSR_MXISA_ZBB`	r/-	`Zbb` ISA Extension available
24	`CSR_MXISA_ZBS`	r/-	`Zbs` ISA Extension available
25	`CSR_MXISA_ZAAMO`	r/-	`Zaamo` ISA Extension available
28:26	-	r/-	*reserved*, hardwired to zero
27	`CSR_MXISA_CLKGATE`	r/-	sleep-mode clock gating implemented when set (`CPU_CLOCK_GATING_EN`), see CPU Tuning Options
28	`CSR_MXISA_RFHWRST`	r/-	full hardware reset of register file available when set (`CPU_RF_HW_RST_EN`), see CPU Tuning Options
29	`CSR_MXISA_FASTMUL`	r/-	fast multiplication available when set (`CPU_FAST_MUL_EN`), see CPU Tuning Options
30	`CSR_MXISA_FASTSHIFT`	r/-	fast shifts available when set (`CPU_FAST_SHIFT_EN`), see CPU Tuning Options
31	`CSR_MXISA_IS_SIM`	r/-	set if CPU is being **simulated**

3.9. Traps, Exceptions and Interrupts

In this document the following terminology is used (derived from the RISC-V trace specification available at https://github.com/riscv-non-isa/riscv-trace-spec):

- **exception**: an unusual condition occurring at run time associated (i.e. *synchronous*) with an instruction in a RISC-V hart
- **interrupt**: an external *asynchronous* event that may cause a RISC-V hart to experience an unexpected transfer of control
- **trap**: the transfer of control to a trap handler caused by either an *exception* or an *interrupt*

Whenever an exception or interrupt is triggered, the CPU switches to machine-mode (if not already in machine-mode) and continues operation at the address being stored in the `mtvec` CSR. The cause of the the trap can be determined via the `mcause` CSR. A list of all implemented `mcause` values and the according description can be found below in section NEORV32 Trap Listing. The address that reflects the current program counter when a trap was taken is stored to `mepc` CSR. Additional information regarding the cause of the trap can be retrieved from the `mtval` and `mtinst` CSRs.

The traps are prioritized. If several *exceptions* occur at once only the one with highest priority is triggered while all remaining exceptions are ignored and discarded. If several *interrupts* trigger at once, the one with highest priority is serviced first while the remaining ones stay *pending*. After completing the interrupt handler the interrupt with the second highest priority will get serviced and so on until no further interrupts are pending.

Interrupts when in User-Mode

If the core is currently operating in less privileged user-mode, interrupts are globally enabled even if `mstatus`.mie is cleared.

Interrupt Signal Requirements - Standard RISC-V Interrupts

All interrupt request signals are **high-active**. Once triggered, a interrupt request line should stay high until it is explicitly acknowledged by a source-specific mechanism (for example by writing to a specific memory-mapped register).

Instruction Atomicity and Forward-Progress

All instructions execute as atomic operations - interrupts can only trigger *between* consecutive instructions. Additionally, if there is a permanent interrupt request, exactly one instruction from the interrupted program will be executed before another interrupt handler can start. This allows program progress even if there are permanent interrupt requests.

3.9.1. Memory Access Exceptions

If a load operation causes any exception, the instruction's destination register is **not written** at all. Furthermore, exceptions caused by a misaligned memory address a physical memory protection fault do not trigger a memory access request at all.

For 32-bit-only instructions (= no `C` extension) the misaligned instruction exception is raised if bit 1 of the fetch address is set (i.e. not on a 32-bit boundary). If the `C` extension is implemented there will **never** be a misaligned instruction exception at all.

3.9.2. Custom Fast Interrupt Request Lines

As a custom extension, the NEORV32 CPU features 16 fast interrupt request (FIRQ) lines via the `firq_i` CPU top entity signals. These interrupts have custom configuration and status flags in the **`mie`** and **`mip`** CSRs and also provide custom trap codes in **`mcause`**. These FIRQs are reserved for NEORV32 processor-internal usage only.

3.9.3. NEORV32 Trap Listing

The following tables show all traps that are currently supported by the NEORV32 CPU. It also shows the prioritization and the CSR side-effects.

FIRQ Mapping

See section NEORV32-Specific Fast Interrupt Requests for the mapping of the FIRQ channels to the according hardware modules.

Table Annotations

The "Prio." column shows the priority of each trap with the highest priority being 1. The "RTE Trap ID" aliases are defined by the NEORV32 core library (the runtime environment *RTE*) and can be used in plain C code when interacting with the pre-defined RTE function. The **`mcause`**, **`mepc`**, **`mtval`** and **`mtinst`** columns show the value being written to the according CSRs when a trap is triggered:

- **I-PC** - address of intercepted instruction (instruction has *not* been executed yet)
- **PC** - address of instruction that caused the trap (instruction has been executed)
- **ADR** - bad data memory access address that caused the trap
- **INS** - the transformed/decompressed instruction word that caused the trap
- **0** - zero

Table 88. NEORV32 Trap Listing

Prio.	`mcause`	RTE Trap ID	Cause	`mepc`	`mtval`	`mtinst`
Exceptions (*synchronous* to instruction execution)						
1	`0x00000001`	`TRAP_CODE_I_ACCESS`	instruction access fault	I-PC	0	INS
2	`0x00000002`	`TRAP_CODE_I_ILLEGAL`	illegal instruction	PC	0	INS
3	`0x00000000`	`TRAP_CODE_I_MISALIGNED`	instruction address misaligned	PC	0	INS

Prio.	mcause	RTE Trap ID	Cause	mepc	mtval	mtinst
4	`0x0000000b`	`TRAP_CODE_MENV_CALL`	environment call from M-mode	PC	0	INS
5	`0x00000008`	`TRAP_CODE_UENV_CALL`	environment call from U-mode	PC	0	INS
6	`0x00000003`	`TRAP_CODE_BREAKPOINT`	software breakpoint / trigger firing	PC	0	INS
7	`0x00000006`	`TRAP_CODE_S_MISALIGNED`	store address misaligned	PC	ADR	INS
8	`0x00000004`	`TRAP_CODE_L_MISALIGNED`	load address misaligned	PC	ADR	INS
9	`0x00000007`	`TRAP_CODE_S_ACCESS`	store access fault	PC	ADR	INS
10	`0x00000005`	`TRAP_CODE_L_ACCESS`	load access fault	PC	ADR	INS
Interrupts (*asynchronous* to instruction execution)						
11	`0x80000010`	`TRAP_CODE_FIRQ_0`	fast interrupt request channel 0	I-PC	0	0
12	`0x80000011`	`TRAP_CODE_FIRQ_1`	fast interrupt request channel 1	I-PC	0	0
13	`0x80000012`	`TRAP_CODE_FIRQ_2`	fast interrupt request channel 2	I-PC	0	0
14	`0x80000013`	`TRAP_CODE_FIRQ_3`	fast interrupt request channel 3	I-PC	0	0
15	`0x80000014`	`TRAP_CODE_FIRQ_4`	fast interrupt request channel 4	I-PC	0	0
16	`0x80000015`	`TRAP_CODE_FIRQ_5`	fast interrupt request channel 5	I-PC	0	0
17	`0x80000016`	`TRAP_CODE_FIRQ_6`	fast interrupt request channel 6	I-PC	0	0
18	`0x80000017`	`TRAP_CODE_FIRQ_7`	fast interrupt request channel 7	I-PC	0	0
19	`0x80000018`	`TRAP_CODE_FIRQ_8`	fast interrupt request channel 8	I-PC	0	0
20	`0x80000019`	`TRAP_CODE_FIRQ_9`	fast interrupt request channel 9	I-PC	0	0
21	`0x8000001a`	`TRAP_CODE_FIRQ_10`	fast interrupt request channel 10	I-PC	0	0
22	`0x8000001b`	`TRAP_CODE_FIRQ_11`	fast interrupt request channel 11	I-PC	0	0
23	`0x8000001c`	`TRAP_CODE_FIRQ_12`	fast interrupt request channel 12	I-PC	0	0
24	`0x8000001d`	`TRAP_CODE_FIRQ_13`	fast interrupt request channel 13	I-PC	0	0
25	`0x8000001e`	`TRAP_CODE_FIRQ_14`	fast interrupt request channel 14	I-PC	0	0
26	`0x8000001f`	`TRAP_CODE_FIRQ_15`	fast interrupt request channel 15	I-PC	0	0
27	`0x8000000b`	`TRAP_CODE_MEI`	machine external interrupt (MEI)	I-PC	0	0
28	`0x80000003`	`TRAP_CODE_MSI`	machine software interrupt (MSI)	I-PC	0	0
29	`0x80000007`	`TRAP_CODE_MTI`	machine timer interrupt (MTI)	I-PC	0	0

Table 89. NEORV32 Trap Description

Copyright by Stephan Nolting. All rights reserved.

Trap ID [C]	Triggered when …
`TRAP_CODE_I_ACCESS`	bus timeout, bus access error or PMP rule violation during instruction fetch
`TRAP_CODE_I_ILLEGAL`	trying to execute an invalid instruction word (malformed or not supported) or on a privilege violation
`TRAP_CODE_I_MISALIGNED`	fetching a 32-bit instruction word that is not 32-bit-aligned (see note below)
`TRAP_CODE_MENV_CALL`	executing `ecall` instruction in machine-mode
`TRAP_CODE_UENV_CALL`	executing `ecall` instruction in user-mode
`TRAP_CODE_BREAKPOINT`	executing `ebreak` instruction or if Trigger Module fires
`TRAP_CODE_S_MISALIGNED`	storing data to an address that is not naturally aligned to the data size (half/word)
`TRAP_CODE_L_MISALIGNED`	loading data from an address that is not naturally aligned to the data size (half/word)
`TRAP_CODE_L_ACCESS`	bus timeout, bus access error or PMP rule violation during load data operation
`TRAP_CODE_S_ACCESS`	bus timeout, bus access error or PMP rule violation during store data operation
`TRAP_CODE_FIRQ_*`	caused by interrupt-condition of **processor-internal modules**, see NEORV32-Specific Fast Interrupt Requests
`TRAP_CODE_MEI`	machine external interrupt (via dedicated Processor Top Entity - Signals)
`TRAP_CODE_MSI`	machine software interrupt (internal Core Local Interruptor (CLINT) or via dedicated Processor Top Entity - Signals)
`TRAP_CODE_MTI`	machine timer interrupt (internal Core Local Interruptor (CLINT) or via dedicated Processor Top Entity - Signals)

Resumable Exceptions

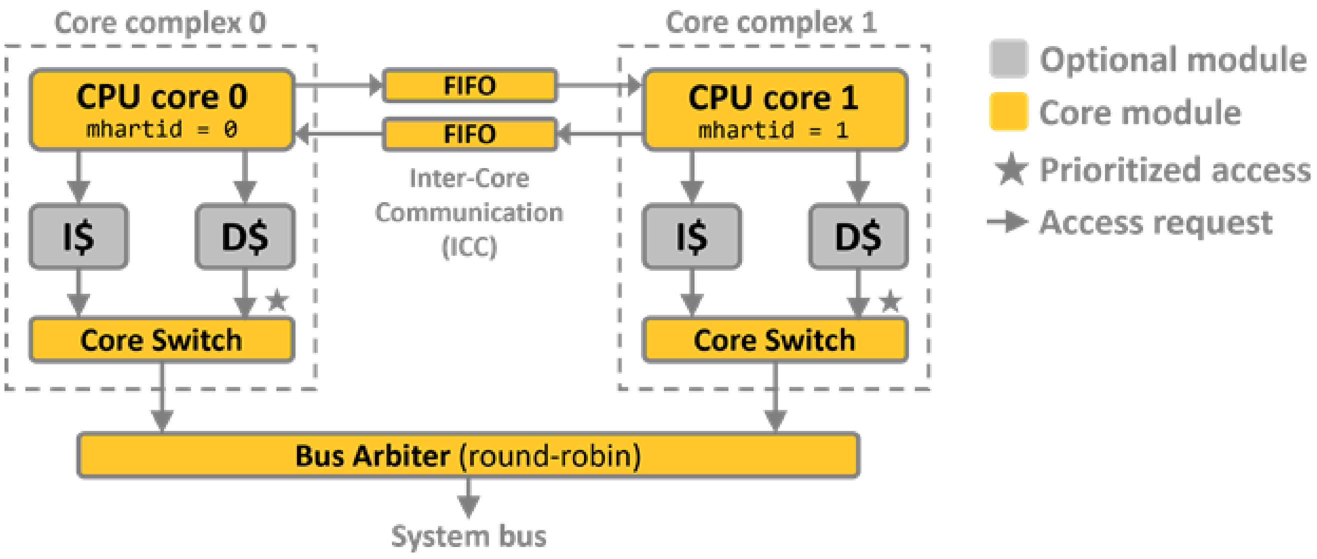
Note that not all exceptions are resumable. For example, the "instruction access fault" exception or the "instruction address misaligned" exception are not resumable in most cases. These exception might indicate a fatal memory hardware failure.

3.10. Dual-Core Configuration

Dual-Core Example Programs

A set of rather simple dual-core example programs can be found in `sw/example/demo_dual_core*`.

Optionally, the CPU core can be implemented as **symmetric multiprocessing (SMP) dual-core** system. This dual-core configuration is enabled by the `DUAL_CORE_EN` top generic. When enabled, two *core complexes* are implemented. Each core complex consists of a CPU core and optional instruction (`I$`) and data (`D$`) caches. Similar to the single-core Bus System, the instruction and data interfaces are switched into a single bus interface by a prioritizing bus switch. The bus interfaces of both core complexes are further switched into a single system bus using a round-robin arbiter.



Both CPU cores are fully identical and use the same ISA, tuning and cache configurations provided by the according top generics. However, each core can be identified by the according "hart ID" that can be retrieved from the **`mhartid`** CSR. CPU core 0 (the *primary* core) has `mhartid = 0` while core 1 (the *secondary* core) has `mhartid = 1`.

The following table summarizes the most important aspects when using the dual-core configuration.

CPU configuration	Both cores use the same cache, CPU and ISA configuration provided by the according top generics.
Debugging	A special SMP openOCD script (`sw/openocd/openocd_neorv32.dual_core.cfg`) is required to debug both cores at once. SMP-debugging is fully supported by the RISC-V gdb port.
Clock and reset	Both cores use the same global processor clock and reset. If CPU Clock Gating is enabled, the clock of each core can be individually halted by putting the core into Sleep Mode.
Address space	Both cores have full access to the same physical Address Space.

Interrupts	All Processor Interrupts are routed to both cores. Hence, each core has access to all NEORV32-Specific Fast Interrupt Requests (FIRQs). Additionally, the RISC-V machine-level *external interrupt* (via the top `mext_irq_i` port) is also send to both cores. In contrast, the RISC-V machine level *software* and *timer* interrupts are core-exclusive (provided by the Core Local Interruptor (CLINT)).
RTE	The NEORV32 Runtime Environment can be used for both cores. However, the RTE needs to be explicitly initialized on each core (executing `neorv32_rte_setup()`). Note that the installed trap handlers apply to both cores. The installed user-defined trap handlers can determine the core's ID to perform core-specific trap handling.
Memory	Each core has its own stack. The top of stack of core 0 is defined by the Linker Script while the top of stack of core 1 has to be explicitly defined by core 0 (see Dual-Core Boot). Both cores share the same heap, `.data` and `.bss` sections. Hence, only core 0 setups the `.data` and `.bss` sections at boot-up.
Constructors and destructors	Constructors and destructors are executed by core 0 only (see section C Standard Library).
Cache coherency	Be aware that there is no cache snooping available. If any level-1 cache is enabled (Processor-Internal Instruction Cache (iCACHE) and/or Processor-Internal Data Cache (dCACHE)) care must be taken to prevent access to outdated data - either by using cache synchronization (`fence` / `fence.i` instructions) or by using Atomic Memory Access.
Inter-core communication	See section Inter-Core Communication (ICC).
Bootloader	Only core 0 will boot and execute the bootloader while core 1 is held in standby.
Booting	See section Dual-Core Boot.

3.10.1. SMP Software Library

An SMP library provides basic functions for launching the secondary core and for performing direct core-to-core communication:

neorv32_smp.c	Online software reference (Doxygen)
neorv32_smp.h	Online software reference (Doxygen)

3.10.2. Inter-Core Communication (ICC)

Both cores can communicate with each other via a direct point-to-point connection based on FIFO-like message queues. These direct communication links are faster (in terms of latency) compared to a memory-mapped or shared-memory communication. Additionally, communication using these links is guaranteed to be atomic.

The inter-core communication (ICC) module is implemented as dedicated hardware module within each CPU core (VHDL file `rtl/core/neorv32_cpu_icc.vhd`). This module is automatically included if the dual-core option is enabled. Each core provides a **32-bit wide** and **4 entries deep** FIFO for sending data to the other core. Hence, there are two FIFOs: one for sending data from core 0 to core 1 and another one for sending data the opposite way.

The ICC communication links are accessed via two NEORV32-specific CSRs. Hence, those FIFOs are accessible only by the CPU core itself and cannot be accessed by the DMA or any other CPU core.

The **`mxiccsreg`** provides read-only status information about the core's ICC links: bit 0 becomes set if there is RX data available for *this* core (send from the the other core). Bit 1 is set as long there is free space in *this* core's TX data FIFO. The **`mxiccdata`** CSR is used for actual data send/receive operations. Writing this register will put the according data word into the TX link FIFO of *this* core. Reading this CSR will return a data word from the RX FIFO of *this* core.

The ICC FIFOs do not provide any interrupt capabilities. Software is expected to use the machine-software interrupt of the receiving core (provided by the Core Local Interruptor (CLINT)) to inform it about available messages.

3.10.3. Dual-Core Boot

After reset, both cores start booting. However, core 1 will - regardless of the Boot Configuration - always enter Sleep Mode right inside the default Start-Up Code (crt0) that is linked with any compiled application. The primary core (core 0) will continue booting, executing either the Bootloader or the pre-installed image from the internal instruction memory (depending on the boot configuration).

To boot-up core 1, the primary core has to use a special library function provided by the NEORV32 software framework:

Listing 15. CPU Core 1 launch function prototype (note that this function can only be executed on core 0)

```
int neorv32_smp_launch(int (*entry_point)(void), uint8_t* stack_memory, size_t
stack_size_bytes);
```

When executed, core 0 uses the Inter-Core Communication (ICC) to send launch data that includes the entry point for core 1 (via `entry_point`) and the actual stack configuration (via `stack_memory` and `stack_size_bytes`). Note that the main function for core 1 has to use a specific type (return `int`, no arguments):

Listing 16. CPU Core 1 Main Function

```
int core1_main(void) {
  return 0; // return to crt0 and go to sleep mode
}
```

Core 1 Stack Memory

The memory for the stack of core 1 (`stack_memory`) can be either statically allocated (i.e. a global volatile memory array; placed in the `.data` or `.bss` section of core 0) or dynamically allocated (using `malloc`; placed on the heap of core 0). In any case the memory should be aligned to a 16-byte boundary.´

After that, the primary core triggers the *machine software interrupt* of core 1 using the Core Local Interruptor (CLINT). Core 1 wakes up from sleep mode, consumes the configuration structure and finally starts executing at the provided entry point. When `neorv32_smp_launch()` returns (with no error code) the secondary core is online and running.

Chapter 4. Software Framework

The NEORV32 project comes with a complete software ecosystem called the "software framework" which is based on the C-language RISC-V GCC port and consists of the following parts:

- Compiler Toolchain
- Core Libraries
- System View Description File (SVD)
- Application Makefile
- Default Compiler Flags
- Linker Script
- C Standard Library
- Start-Up Code (crt0)
- Executable Image Formats
- NEORV32 Runtime Environment
- Bootloader

Software Documentation

All core libraries and example programs are documented "in-code" using **Doxygen**. The documentation is automatically built and deployed to GitHub pages and is available online at https://stnolting.github.io/neorv32/sw/files.html.

Example Programs

A collection of annotated example programs illustrating how to use certain CPU functions and peripheral/IO modules can be found in `sw/example`.

4.1. Compiler Toolchain

The toolchain for this project is based on the free and open RISC-V GCC-port. You can find the compiler sources and build instructions in the official RISC-V GNU toolchain GitHub repository: https://github.com/riscv/riscv-gnutoolchain.

Toolchain Installation

More information regarding the toolchain (building from scratch or downloading prebuilt ones) can be found in the user guide section Software Toolchain Setup.

4.2. Core Libraries

The NEORV32 project provides a set of pre-defined C libraries that allow an easy integration of the processor/CPU features (also called "HAL" - *hardware abstraction layer*). All driver and runtime-related files are located in `sw/lib`. These library files are automatically included and linked by adding the following include statement:

```
#include <neorv32.h> // NEORV32 HAL, core and runtime libraries
```

The NEORV32 HAL consists of the following files.

Table 90. NEORV32 Hardware Abstraction Layer File List

C source file	C header file	Description
-	`neorv32.h`	Main NEORV32 library file
`neorv32_aux.c`	`neorv32_aux.h`	General auxiliary/helper function
`neorv32_cfs.c`	`neorv32_cfs.h`	Custom Functions Subsystem (CFS) HAL
`neorv32_clint.c`	`neorv32_clint.h`	Core Local Interruptor (CLINT) HAL
`neorv32_cpu.c`	`neorv32_cpu.h`	NEORV32 Central Processing Unit (CPU) HAL
	`neorv32_cpu_csr.h`	Control and Status Registers (CSRs) definitions
`neorv32_cpu_cfu.c`	`neorv32_cpu_cfu.h`	Custom Functions Unit (CFU) HAL
`neorv32_crc.c`	`neorv32_crc.h`	Cyclic Redundancy Check (CRC) HAL
`neorv32_dma.c`	`neorv32_dma.h`	Direct Memory Access Controller (DMA) HAL
`neorv32_gpio.c`	`neorv32_gpio.h`	General Purpose Input and Output Port (GPIO) HAL
`neorv32_gptmr.c`	`neorv32_gptmr.h`	General Purpose Timer (GPTMR) HAL
-	`neorv32_intrinsics.h`	Macros for intrinsics and custom instructions
`neorv32_neoled.c`	`neorv32_neoled.h`	Smart LED Interface (NEOLED) HAL
`neorv32_onewire.c`	`neorv32_onewire.h`	One-Wire Serial Interface Controller (ONEWIRE) HAL
`neorv32_pwm.c`	`neorv32_pwm.h`	Pulse-Width Modulation Controller (PWM) HAL
`neorv32_rte.c`	`neorv32_rte.h`	NEORV32 Runtime Environment
`neorv32_sdi.c`	`neorv32_sdi.h`	Serial Data Interface Controller (SDI) HAL
`neorv32_slink.c`	`neorv32_slink.h`	Stream Link Interface (SLINK) HAL
`neorv32_smp.c`	`neorv32_smp.h`	HAL for the SMP Dual-Core Configuration
`neorv32_spi.c`	`neorv32_spi.h`	Serial Peripheral Interface Controller (SPI) HAL

C source file	C header file	Description
	`neorv32_sysinfo.h`	System Configuration Information Memory (SYSINFO) HAL
`neorv32_trng.c`	`neorv32_trng.h`	True Random-Number Generator (TRNG) HAL
`neorv32_twd.c`	`neorv32_twd.h`	Two-Wire Serial Device Controller (TWD) HAL
`neorv32_twi.c`	`neorv32_twi.h`	Two-Wire Serial Interface Controller (TWI) HAL
`neorv32_uart.c`	`neorv32_uart.h`	Primary Universal Asynchronous Receiver and Transmitter (UART0) and UART1 HAL
`neorv32_wdt.c`	`neorv32_wdt.h`	Watchdog Timer (WDT) HAL
`neorv32_xip.c`	`neorv32_xip.h`	Execute In Place Module (XIP) HAL
`neorv32_newlib.c`	-	Platform-specific system calls for *newlib*

Core Libraries Documentation

The Doxygen-based documentation of the software framework including all core libraries is available online at https://stnolting.github.io/neorv32/sw/files.html.

4.3. System View Description File (SVD)

A CMSIS-SVD-compatible **System View Description (SVD)** file including all peripherals is available in `sw/svd`.

4.4. Application Makefile

Application compilation is based on a centralized GNU makefile (`sw/common/common.mk`). Each software project (for example the ones in `sw/example` folder) should provide a local makefile that just includes the central makefile:

```
# Set path to NEORV32 root directory
NEORV32_HOME ?= ../../..
# Include the main NEORV32 makefile
include $(NEORV32_HOME)/sw/common/common.mk
```

Thus, the functionality of the central makefile (including all targets) becomes available for the project. The project-local makefile should be used to define all setup-relevant configuration options instead of changing the central makefile to keep the code base clean. Setting variables in the project-local makefile will override the default configuration. Most example projects already provide a makefile that list all relevant configuration options.

The following example shows all relevant configuration variables:

```
# Override the default CPU ISA
MARCH = rv32imc_zicsr_zifencei

# Override the default RISC-V GCC prefix
RISCV_PREFIX ?= riscv-none-elf-

# Override default optimization goal
EFFORT = -Os

# Add extended debug symbols
USER_FLAGS += -ggdb -gdwarf-3

# Additional sources
APP_SRC += $(wildcard ./*.c)
APP_INC += -I .

# Adjust processor IMEM size
USER_FLAGS += -Wl,--defsym,__neorv32_rom_size=16k

# Adjust processor DMEM size
USER_FLAGS += -Wl,--defsym,__neorv32_ram_size=8k

# Adjust maximum heap size
USER_FLAGS += -Wl,--defsym,__neorv32_heap_size=1k

# Reduce library footprint when no UART is synthesized
#USER_FLAGS += -DUART_DISABLED
```

```
# Enable link-time-optimization
#USER_FLAGS += -flto

# Additional compiler flags (append to this variable)
#USER_FLAGS += ...

# Set path to NEORV32 root directory
NEORV32_HOME ?= ../../..

# Include the main NEORV32 makefile
include $(NEORV32_HOME)/sw/common/common.mk
```

Setup of a New Project

When creating a new project, copy an existing project folder or at least the makefile to the new project folder. It is recommended to create new projects also in `sw/example` to keep the file dependencies. However, these dependencies can be manually configured via makefile variables if the new project is located somewhere else. For more complex projects, it may be useful to use explicit `source` and `include` folders. See `sw/example/coremark` for an example.

4.4.1. Makefile Targets

Invoking a project-local makefile (executing `make` or `make help`) will show the help menu that lists all available targets as well as all variable including their *current* setting.

```
neorv32/sw/example/hello_world$ make
NEORV32 Software Makefile
Find more information at https://github.com/stnolting/neorv32

Targets:

  help         - show this text
  check        - check toolchain
  info         - show makefile/toolchain configuration
  gdb          - start GNU debugging session
  asm          - compile and generate <main.asm> assembly listing file for manual
debugging
  elf          - compile and generate <main.elf> ELF file
  exe          - compile and generate <neorv32_exe.bin> executable image file for
bootloader upload (includes a HEADER!)
  bin          - compile and generate <neorv32_raw_exe.bin> executable memory image
  hex          - compile and generate <neorv32_raw_exe.hex> executable memory image
  coe          - compile and generate <neorv32_raw_exe.coe> executable memory image
  mem          - compile and generate <neorv32_raw_exe.mem> executable memory image
  mif          - compile and generate <neorv32_raw_exe.mif> executable memory image
  image        - compile and generate VHDL IMEM application boot image
<neorv32_application_image.vhd> in local folder
```

```
  install      - compile, generate and install VHDL IMEM application boot image
<neorv32_application_image.vhd>
  sim          - in-console simulation using default/simple testbench and GHDL
  hdl_lists    - regenerate HDL file-lists (*.f) in NEORV32_HOME/rtl
  all          - exe + install + hex + bin + asm
  elf_info     - show ELF layout info
  elf_sections - show ELF sections
  clean        - clean up project home folder
  clean_all    - clean up project home folder and image generator
  bl_image     - compile and generate VHDL BOOTROM bootloader boot image
<neorv32_bootloader_image.vhd> in local folder
  bootloader   - compile, generate and install VHDL BOOTROM bootloader boot image
<neorv32_bootloader_image.vhd>

Variables:

  USER_FLAGS     - Custom toolchain flags [append only]: "-ggdb -gdwarf-3 -Wl,
--defsym,__neorv32_rom_size=16k -Wl,--defsym,__neorv32_ram_size=8k"
  USER_LIBS      - Custom libraries [append only]: ""
  EFFORT         - Optimization level: "-Os"
  MARCH          - Machine architecture: "rv32i_zicsr_zifencei"
  MABI           - Machine binary interface: "ilp32"
  APP_INC        - C include folder(s) [append only]: "-I ."
  APP_SRC        - C source folder(s) [append only]: "./main.c   "
  ASM_INC        - ASM include folder(s) [append only]: "-I ."
  RISCV_PREFIX   - Toolchain prefix: "riscv-none-elf-"
  NEORV32_HOME   - NEORV32 home folder: "../../.."
  GDB_ARGS       - GDB (connection) arguments: "-ex target extended-remote
localhost:3333"
  GHDL_RUN_FLAGS - GHDL simulation run arguments: ""
```

Build Artifacts

All *intermediate* build artifacts (like object files and binaries) will be places into a (new) project-local folder named `build`. The *resulting* build artifacts (like executable, the main ELF and all memory initialization/image files) will be placed in the root project folder.

4.4.2. Default Compiler Flags

The central makefile uses specific compiler flags to tune the code to the NEORV32 hardware. Hence, these flags should not be altered. However, experienced users can modify them to further tune compilation.

Table 91. Compiler Options (`CC_OPTS`)

`-Wall`	Enable all compiler warnings.

Copyright by Stephan Nolting. All rights reserved.

`-ffunction-sections`	Put functions in independent sections. This allows a code optimization as dead code can be easily removed.
`-fdata-sections`	Put data segment in independent sections. This allows a code optimization as unused data can be easily removed.
`-nostartfiles`	Do not use the default start code. Instead, the NEORV32-specific start-up code (`sw/common/crt0.S`) is used (pulled-in by the linker script).
`-mno-fdiv`	Use built-in software functions for floating-point divisions and square roots (since the according instructions are not supported yet).
`-mstrict-align`	Unaligned memory accesses cannot be resolved by the hardware and require emulation.
`-mbranch-cost=10`	Branching costs a lot of cycles.
`-Wl,--gc-sections`	Make the linker perform dead code elimination.
`-ffp-contract=off`	Disable floating-point expression contraction.
`-g`	Add (simple) debug information.

Checking Compiler Flags from a Compiled Program

The makefile's `CC_OPTS` is exported as **define** to be available within a C program; for example `neorv32_uart0_printf("%s\n", CC_OPTS);`.

Table 92. Linker Libraries (`LD_LIBS`)

`-lm`	Include/link with `math.h`.
`-lc`	Search for the standard C library when linking.
`-lgcc`	Make sure we have no unresolved references to internal GCC library subroutines.

Advanced Debug Symbols

By default, only "simple" symbols are added to the ELF (`-g`). Extended debug flags (e.g. for Eclipse) can be added using the `USER_FLAGS` variable (e.g. `USER_FLAGS += -ggdb -gdwarf-3`). Note that other debug flags may be required depending of the GCC/GDB version

4.5. Linker Script

The NEORV32-specific linker script (`sw/common/neorv32.ld`) is used to link the compiled sources according to the processor's Address Space). For the final executable, only two memory segments are required:

Table 93. Linker script - Memory Segments

Memory section	Description
`rom`	Instruction memory address space (processor-internal Instruction Memory (IMEM) and/or external memory)
`ram`	Data memory address space (processor-internal Data Memory (DMEM) and/or external memory)

These two sections are configured by several variables defined in the linker script and exposed to the build framework (aka the makefile). Those variable allow to customized the RAM/ROM sizes and base addresses. Additionally, a certain amount of the RAM can be reserved for the software-managed heap (see RAM Layout).

Table 94. Linker script - Configuration

Memory section	Description	Default
`__neorv32_rom_size`	"ROM" size (instruction memory / IMEM)	16kB
`__neorv32_ram_size`	"RAM" size (data memory / DMEM)	8kB
`__neorv32_rom_base`	"ROM" base address (instruction memory / IMEM)	`0x00000000`
`__neorv32_ram_base`	"RAM" base address (data memory / DMEM)	`0x80000000`
`__neorv32_heap_size`	Maximum heap size; part of the "RAM"	0kB

Each variable provides a default value (e.g. "16K" for the instruction memory /ROM /IMEM size). These defaults can be overridden by setup-specific values to take the user-defined processor configuration into account (e.g. a different IMEM size). The `USER_FLAGS` variable provided by the Application Makefile can also be used to customize the memory configuration. For example, the following line can be added to a project-specific local makefile to adjust the memory sizes:

Listing 17. Overriding Default Memory Sizes (configuring 64kB IMEM and 32kB DMEM)

```
USER_FLAGS += "-Wl,--defsym,__neorv32_rom_size=64k -Wl,
--defsym,__neorv32_ram_size=32k"
```

Memory Configuration Constraints

Memory sizes have to be a multiple of 4 bytes. Memory base addresses have to be 32-bit-aligned.

4.5.1. RAM Layout

The default NEORV32 linker script uses the defined RAM size to map several sections. Note that depending on the application some sections might have zero size.

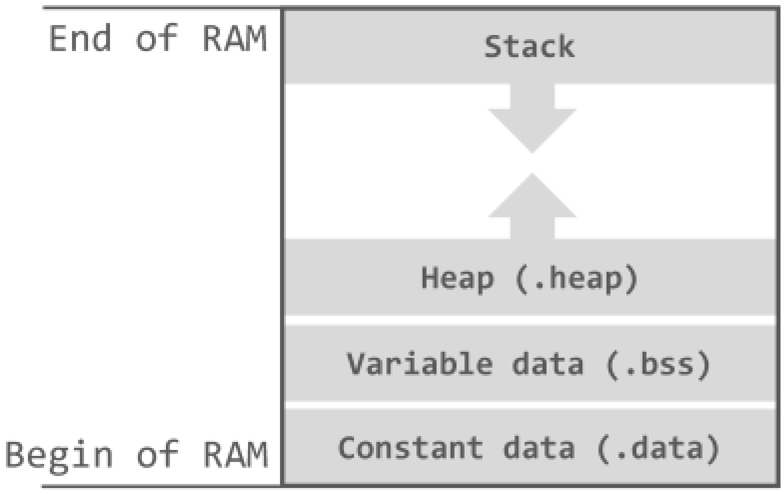


Figure 12. Default RAM Layout

1. **Constant data (`.data`)**: The constant data section is placed right at the beginning of the RAM. For example, this section contains *explicitly initialized* global variables. This section is initialized by the Start-Up Code (crt0).
2. **Dynamic data (`.bss`)**: The constant data section is followed by the dynamic data section that contains *uninitialized* data like global variables without explicit initialization. This section is cleared by the Start-Up Code (crt0).
3. **Heap (`.heap`)**: The heap is used for dynamic memory that is managed by functions like `malloc()` and `free()`. The heap grows upwards. This section is not initialized at all.
4. **Stack**: The stack starts at the end of the RAM at the last 16-byte aligned address. According to the RISC-V ABI / calling convention the stack is 128-bit-aligned before procedure entry. The stack grows downwards.

Heap Size

The maximum size of the heap is defined by the `__neorv32_heap_size` variable. This variable has to be **explicitly defined** in order to define a heap size (and to use dynamic memory allocation at all) other than zero.

Heap-Stack Collision

Take care when using dynamic memory to avoid collision of the heap and stack memory areas. There is no compile-time protection mechanism available as the actual heap and stack size are defined by *runtime* data.

4.6. C Standard Library

The default software framework relies on **newlib** as default C standard library. Newlib provides hooks for common "system calls" (like file handling and standard input/output) that are used by other C libraries like `stdio`. These hooks are available in `sw/lib/source/newlib.c` and were adapted for the NEORV32 processor.

Standard Consoles

The UART0 is used to implement all the standard input, output and error consoles (`STDIN`, `STDOUT` and `STDERR`). Note that `\n` (newline) is automatically converted to `\r\n` (carriage-return and newline).

Constructors and Destructors

Constructors and destructors for plain C code or for C++ applications are supported by the software framework. See `sw/example/hello_cpp` for a minimal example. Note that constructor and destructors are only executed by core 0 (primary core) in the SMP Dual-Core Configuration.

Newlib Test/Demo Program

A simple test and demo program that uses some of newlib's system functions (like `malloc`/`free` and `read`/`write`) is available in `sw/example/demo_newlib`.

4.7. Start-Up Code (crt0)

The CPU and also the processor require a minimal start-up and initialization code to bring the hardware into an operational state. Furthermore, the C runtime requires an initialization before compiled code can be executed. This setup is done by the start-up code (`sw/common/crt0.S`) which is automatically linked with *every* application program and gets mapped before the actual application code so it gets executed right after boot.

The `crt0.S` start-up performs the following operations:

1. Setup the stack pointer and the global pointer according to the RAM Layout provided by the Linker Script symbols.
2. Initialize **`mstatus`** CSR disabling machine-level interrupts.
3. Install an Early Trap Handler to **`mtvec`** CSR.
4. Clear **`mie`** CSR disabling all interrupt sources.
5. Initialize all integer register `x1` - `x31` (only `x1` - `x15` if the `E` CPU extension is enabled).
6. If the executing CPU core is not core 0 an SMP-specific code is executed and the CPU is halted in sleep mode. See section Dual-Core Boot for more information.
7. Setup `.data` section to configure initialized variables.
8. Clear the `.bss` section.
9. Call all *constructors* (if there are any).
10. Call the application's `main()` function (with no arguments; `argc` = `argv` = 0).
11. If `main()` returns:
    - All interrupt sources are disabled by clearing **`mie`** CSR.
    - The return value of `main()` is copied to the **`mscratch`** CSR to allow inspection by the debugger.
    - The Early Trap Handler is re-installed to **`mtvec`** CSR.
    - Call all *destructors* (if there are any).
    - Execute an `ebreak` instruction to enter debug mode if an external debugger is connected.
    - The CPU enters sleep mode executing the `wfi` instruction in an endless loop.

4.7.1. Early Trap Handler

The start-up code provides a very basic trap handler for the early boot phase. This handler does nothing but trying to move on to the next linear instruction whenever an interrupt or synchronous exception is encountered. This simple trap handler does not interact with the stack at all as it just uses a single register that is backup-ed using the **`mscratch`** CSR.

4.8. Executable Image Formats

The compiled and linked executable (ELF file) is further processed by the NEORV32 image generator (`sw/image_gen`) to generate the final executable file. The image generator can generate several types of executable file formats selected by a flag when calling the generator. **Note that all these options are managed by the makefile (see Makefile Targets).**

`-app_bin`	Generates an executable binary file (including a bootloader header) for upload via the bootloader.
`-app_vhd`	Generates an executable VHDL memory initialization image for the processor-internal IMEM.
`-bld_vhd`	Generates an executable VHDL memory initialization image for the processor-internal BOOT ROM.
`-raw_hex`	Generates a raw 8x ASCII hex-char file for custom purpose.
`-raw_bin`	Generates a raw binary file `for custom purpose.
`-raw_coe`	Generates a raw COE file for FPGA memory initialization.
`-raw_mem`	Generates a raw MEM file for FPGA memory initialization.
`-raw_mif`	Generates a raw MIF file for FPGA memory initialization.

Image Generator Compilation

The sources of the image generator are automatically compiled when invoking the makefile (requiring a *native* GCC installation).

Executable Header

for the `app_bin` option the image generator adds a small header to the executable. This header is required by the Bootloader to identify and manage the executable. The header consists of three 32-bit words located right at the beginning of the file. The first word of the executable is the signature word and is always `0x4788cafe`. Based on this word the bootloader can identify a valid image file. The next word represents the size in bytes of the actual program image in bytes. A simple complement checksum of the actual program image is given by the third word. This provides a simple protection against data transmission or storage errors. **Note that this executable format cannot be used for *direct* execution (e.g. via XIP or direct memory access).**

4.9. Bootloader

Pre-Built Bootloader Image

This section refers to the **default** NEORV32 bootloader. A pre-compiled memory image for the processor-internal Bootloader ROM (BOOTROM) is available in the project's rtl folder: `rtl/core/neorv32_bootloader_image.vhd`. This image is automatically inserted into the boot ROM when synthesizing the processor with the bootloader being enabled.

Minimal RISC-V ISA and Memory Configuration

The default bootloader image was compiled for a minimal `rv32e_zicsr_zifencei` ISA configuration and only requires a RAM size of at least 256 bytes. Both constraints ensure that the bootloader can be executed by any actual CPU/processor configuration. However, the bootloader can recompiled with different capabilities. See the User Guide https://stnolting.github.io/neorv32/ug/#_customizing_the_internal_bootloader for more information.

SMP Dual-Core Configuration

For the SMP Dual-Core Configuration only the primary core (core 0) will boot and execute the bootloader while the secondary core (core 1) will be halted in sleep mode.

The NEORV32 bootloader (`sw/bootloader/bootloader.c`) provides an optional built-in firmware that allows to upload new application executables at *any time* without the need to re-synthesize the FPGA's bitstream. A UART connection is used to provide a simple text-based user interface that allows to upload executables.

Furthermore, the bootloader provides options to store an executable to a processor-external SPI flash. An "auto boot" feature can optionally fetch this executable right after reset if there is no user interaction via UART. This allows to build processor setups with *non-volatile application storage* while maintaining the option to update the application software at any timer.

Software Documentation

The Doxygen-based documentation of the bootloader's software is available online: https://stnolting.github.io/neorv32/sw/bootloader_8c.html

4.9.1. Bootloader SoC/CPU Requirements

The bootloader requires certain CPU and SoC extensions and modules to be enabled in order to operate correctly.

REQUIRED	The Boot Configuration (`BOOT_MODE_SELECT` generic) has to be set to "bootloader" mode.

REQUIRED	The bootloader requires the privileged architecture CPU extension (`Zicsr` ISA Extension) to be enabled.
REQUIRED	At least 512 bytes of data memory (processor-internal DMEM or processor-external DMEM) are required for the bootloader's stack and global variables.
RECOMMENDED	For user interaction via the Bootloader Console (like uploading executables) the primary UART (Primary Universal Asynchronous Receiver and Transmitter (UART0)) is required.
RECOMMENDED	The default bootloader uses bit 0 of the General Purpose Input and Output Port (GPIO) output port to drive a high-active "heart beat" status LED.
RECOMMENDED	The machine timer of the Core Local Interruptor (CLINT) is used to control blinking of the status LED and also to automatically trigger the Auto Boot Sequence.
OPTIONAL	The SPI controller (Serial Peripheral Interface Controller (SPI)) is needed to store/load executable from external flash using the Auto Boot Sequence.
OPTIONAL	The XIP controller (Execute In Place Module (XIP)) is needed to boot/execute code directly from a pre-programmed SPI flash.
OPTIONAL	The TWI controller (Two-Wire Serial Interface Controller (TWI)) is needed to boot/execute code directly from pre-programmed TWI memory.

4.9.2. Bootloader Flash Requirements

The bootloader can access an SPI-compatible flash via the processor's top entity SPI port. By default, the flash chip-select line is driven by `spi_csn_o(0)` and the SPI clock uses 1/8 of the processor's main clock as clock frequency. The SPI flash has to support single-byte read and write operations, 24-bit addresses and at least the following standard commands:

- `0x02`: Program page (write byte)
- `0x03`: Read data (byte)
- `0x04`: Write disable (for volatile status register)
- `0x05`: Read (first) status register
- `0x06`: Write enable (for volatile status register)
- `0xAB`: Wake-up from sleep mode (optional)
- `0xD8`: Block erase (64kB)

Custom Configuration

Most properties (like chip select line, flash address width, SPI clock frequency, ...) of the default bootloader can be reconfigured without the need to change the source code. Custom configuration can be made using command line switches (defines) when recompiling the bootloader. See the User Guide https://stnolting.github.io/neorv32/ug/#_customizing_the_internal_bootloader for more information.

4.9.3. Bootloader TWI memory Requirements

The bootloader can access an TWI-compatible memory via the processor's top entity TWI port. Single- and dual address memory is supported, and reading is done in the following pattern `Device Address + Enabled Read | Memory Address Byte 0 | Memory Address 1 (optional) | Read Byte 0 | Read Byte 1 | Read Byte 2 | Read Byte 3`. The addresses are incremented until the end of the program binary is reached.

A python upload script for uploading is provided in the `sw/eeprom_upload` folder. Currently only for the USB-ISS module.

Clock speed information can be read here: Two-Wire Serial Interface Controller (TWI).

4.9.4. Bootloader Console

To interact with the bootloader, connect the primary UART (UART0) signals (`uart0_txd_o` and `uart0_rxd_o`) of the processor's top entity via a serial port (-adapter) to your computer (hardware flow control is not used so the according interface signals can be ignored), configure your terminal program using the following settings and perform a reset of the processor.

Terminal console settings (`19200-8-N-1`):

- 19200 Baud
- 8 data bits
- no parity bit
- 1 stop bit
- newline on `\r\n` (carriage return, newline)
- no transfer protocol / control flow protocol - just raw bytes

Terminal Program

Any terminal program that can connect to a serial port should work. However, make sure the program can transfer data in *raw* byte mode without any protocol overhead (e.g. XMODEM). Some terminal programs struggle with transmitting files larger than 4kB (see https://github.com/stnolting/neorv32/pull/215). Try a different terminal program if uploading of a binary does not work.

The bootloader uses the LSB of the top entity's `gpio_o` output port as high-active status LED. All other output pins are set to low level and won't be altered. After reset, the status LED will start blinking at 2Hz and the following intro screen shows up:

```
<< NEORV32 Bootloader >>

BLDV: Mar  7 2023
```

```
HWV:  0x01080107
CLK:  0x05f5e100
MISA: 0x40901106
XISA: 0xc0000fab
SOC:  0xffff402f
IMEM: 0x00008000
DMEM: 0x00002000

Autoboot in 10s. Press any key to abort.
```

The start-up screen gives some brief information about the bootloader and several system configuration parameters:

`BLDV`	Bootloader version (built date).
`HWV`	Processor hardware version (the **`mimpid`** CSR); in BCD format; example: `0x01040606` = v1.4.6.6).
`CLK`	Processor clock speed in Hz (via the `CLK` register from the System Configuration Information Memory (SYSINFO).
`MISA`	RISC-V CPU extensions (**`misa`** CSR).
`XISA`	NEORV32-specific CPU extensions (**`mxisa`** CSR).
`SOC`	Processor configuration (via the `SOC` register from the System Configuration Information Memory (SYSINFO).
`IMEM`	Internal IMEM size in byte (via the `MEM` register from the System Configuration Information Memory (SYSINFO).
`DMEM`	Internal DMEM size in byte (via the `MEM` register from the System Configuration Information Memory (SYSINFO).

Now you have 10 seconds to press *any* key. Otherwise, the bootloader starts the Auto Boot Sequence. When you press any key within the 10 seconds, the actual bootloader user console starts:

```
<< NEORV32 Bootloader >>

BLDV: Mar  7 2023
HWV:  0x01080107
CLK:  0x05f5e100
MISA: 0x40901106
XISA: 0xc0000fab
SOC:  0xffff402f
IMEM: 0x00008000
DMEM: 0x00002000

Autoboot in 10s. Press any key to abort. ①
Aborted.
```

```
Available CMDs:
 h: Help
 r: Restart
 u: Upload
 s: Store to flash
 l: Load from flash
 t: Load from TWI Device
 x: Boot from flash (XIP)
 e: Execute
CMD:>
```

① Auto boot sequence aborted due to user console input.

The auto boot countdown is stopped and the bootloader's user console is ready to receive one of the following commands:

- `h`: Show the help text (again)
- `r`: Restart the bootloader and the auto-boot sequence
- `u`: Upload new program executable (`neorv32_exe.bin`) via UART into the instruction memory
- `s`: Store executable to SPI flash at `spi_csn_o(0)` (little-endian byte order)
- `l`: Load executable from SPI flash at `spi_csn_o(0)` (little-endian byte order)
- `t`: Load executable from TWI memory at `0x50` (little-endian byte order) (disabled by default)
- `x`: Boot program directly from flash via XIP (requires a pre-programmed image)
- `e`: Start the application, which is currently stored in the instruction memory (IMEM)

A new executable can be uploaded via UART by executing the `u` command. After that, the executable can be directly executed via the `e` command. To store the recently uploaded executable to an attached SPI flash press `s`. To directly load an executable from the SPI flash press `l`. The bootloader and the auto-boot sequence can be manually restarted via the `r` command.

Executable Upload

Make sure to upload the NEORV32 executable `neorv32_exe.bin`. Uploading any other file (like `main.bin`) will cause an `ERR_EXE` bootloader error (see Bootloader Error Codes).

Booting via XIP

The bootloader allows to execute an application right from flash using the Execute In Place Module (XIP) module. This requires a pre-programmed flash. The bootloader's "store" option can **not** be used to program an XIP image.

SPI Flash Power Down Mode

The bootloader will issue a "wake-up" command prior to using the SPI flash to ensure it is not in sleep mode / power-down mode (see https://github.com/stnolting/

neorv32/pull/552).

Default Configuration

More information regarding the default SPI, GPIO, XIP, etc. configuration can be found in the User Guide section https://stnolting.github.io/neorv32/ug/#_customizing_the_internal_bootloader.

SPI Flash Programming

For detailed information on using an SPI flash for application storage see User Guide section Programming an External SPI Flash via the Bootloader.

4.9.5. Auto Boot Sequence

When you reset the NEORV32 processor, the bootloader waits 8 seconds for a UART console input before it starts the automatic boot sequence. This sequence tries to fetch a valid boot image from the external SPI flash, connected to SPI chip select `spi_csn_o(0)` or from external TWI memory. If both are enabled, the bootloader will select SPI. If a valid boot image is found that can be successfully transferred into the instruction memory, it is automatically started. If no SPI flash is detected or if there is no valid boot image found, and error code will be shown.

4.9.6. Bootloader Error Codes

If something goes wrong during bootloader operation an error code and a short message is shown. In this case the processor is halted (entering Sleep Mode), the bootloader status LED is permanently activated and the processor has to be reset manually.

Debugging Information

If an unexpected exception has been raised, the bootloader prints hexadecimal debug information showing the `mcause`, `mepc` and `mtval` CSR values.

`ERR_EXE`	If you try to transfer an invalid executable (via UART or from the external SPI flash), this error message shows up. There might be a transfer protocol configuration error in the terminal program or maybe just the wrong file was selected. Also, if no SPI flash was found during an auto-boot attempt, this message will be displayed.
`ERR_SIZE`	Your program is way too big for the internal processor's instructions memory. Increase the memory size or reduce your application code.
`ERR_CHKS`	This indicates a checksum error. Something went wrong during the transfer of the program image (upload via UART or loading from the external SPI flash). If the error was caused by a UART upload, just try it again. When the error was generated during a flash access, the stored image might be corrupted.

`ERR_FLSH`	This error occurs if the attached SPI flash cannot be accessed. Make sure you have the right type of flash and that it is properly connected to the NEORV32 SPI port using chip select #0.
`ERR_EXC`	The bootloader encountered an unexpected exception during operation. This might be caused when it tries to access peripherals that were not implemented during synthesis. Example: executing commands `l` or `s` (SPI flash operations) without the SPI module being implemented.
`ERR_TWI`	The TWI received an unexpected NACK while reading the external memory. Are the address and speed settings correct?

4.10. NEORV32 Runtime Environment

The NEORV32 software framework provides a minimal runtime environment ("RTE") that takes care of a stable and *safe* execution environment by providing a unified interface for handling of *all* traps (exceptions and interrupts). Once initialized, the RTE provides Default RTE Trap Handlers that catch all possible traps. These default handlers just output a message via UART when a certain trap has been triggered. The default handlers can be overridden by the application code to install application-specific handler functions for each trap.

Using the RTE is **optional but highly recommended** for bare-metal / non-OS applications. The RTE provides a simple and comfortable way of delegating traps to application-specific handlers while making sure that all traps (even though they are not explicitly used by the application) are handled correctly. Performance-optimized applications or embedded operating systems may not use the RTE at all in order to increase response time.

4.10.1. RTE Operation

The RTE manages the trap-related CSRs of the CPU's privileged architecture (see Machine Trap Handling CSRs). It initializes the `mtvec` CSR in DIRECT mode, which provides the base entry point for *all* traps. The address stored to this register defines the address of the **first-level trap handler**, which is provided by the NEORV32 RTE. Whenever an exception or interrupt is triggered this first-level trap handler is executed.

The first-level handler performs a complete context save, analyzes the source of the trap and calls the according **second-level trap handler**, which takes care of the actual exception/interrupt handling. The RTE manages an internal look-up table to track the addresses of the according second-level trap handlers.

After the initial RTE setup, each entry in the RTE's trap handler look-up table is initialized with a Default RTE Trap Handlers. These default handler do not execute any trap-related operations - they just output a debugging message via the primary UART (UART0) (if enabled) to inform the user that a trap has occurred that is not (yet) handled by a proper application-specific trap handler. After sending this message, the RTE tries to resume normal execution by moving on to the next linear instruction.

Dual-Core Configuration

The RTE's internal trap handler look-up table is used globally for **both** cores. If a core-specific handling is required, the according user-defined trap handler need to retrieve the core's ID from `mhartid` and branch accordingly.

4.10.2. Using the RTE

The NEORV32 runtime environment is part of the default NEORV32 software framework. The links to the according software references are listed below.

neorv32_rte.c	Online software reference (Doxygen)
neorv32_rte.h	Online software reference (Doxygen)

The RTE has to be explicitly enabled by calling the according setup function. It is recommended to do this right at the beginning of the application's `main` function. For the SMP Dual-Core Configuration the RTE setup functions has to be called on each core that wants to use the RTE.

Listing 18. RTE Setup Right at the Beginning of "main"

```
int main() {

  neorv32_rte_setup(); // setup NEORV32 runtime environment

  ...
```

After setup, all traps will trigger execution of the RTE's Default RTE Trap Handlers at first. In order to use application-specific trap handlers the default debug handlers can be overridden by installing user-defined ones:

Listing 19. Installing an Application-Specific Trap Handler (Function Prototype)

```
int neorv32_rte_handler_install(uint8_t id, void (*handler)(void));
```

The first argument `id` defines the "trap ID" (for example a certain interrupt request) that shall be handled by the user-defined handler. These IDs are defined in `sw/lib/include/neorv32_rte.h`. However, more convenient device-specific aliases are also defined in `sw/lib/include/neorv32.h`. The second argument `handler` is the actual function that implements the user-defined trap handler. The custom handler functions must have a specific type without any arguments and with no return value:

Listing 20. Custom Trap Handler (Function Prototype)

```
void custom_trap_handler_xyz(void) {

  // handle trap...
}
```

Custom Trap Handler Attributes

Do **NOT** use the `interrupt` attribute for the application trap handler functions! This would place an `mret` instruction at the end of the handler making it impossible to return to the first-level trap handler of the RTE core.

`mscratch` *CSR*

> The `mscratch` CSR should not be used inside an application trap handler as this register is used by the RTE to provide the base address of the application's stack frame Application Context Handling (i.e. modifying the registers of application code that caused a trap).

The following example shows how to install trap handlers for exemplary traps.

Listing 21. Installing Custom Trap Handlers Examples

```
neorv32_rte_handler_install(RTE_TRAP_MTI, machine_timer_irq_handler); // handler for
machine timer interrupt
neorv32_rte_handler_install(RTE_TRAP_MENV_CALL, environment_call_handler); // handler
for machine environment call exception
neorv32_rte_handler_install(SLINK_RX_RTE_ID, slink_rx_handler); // handler for SLINK
receive interrupt
```

4.10.3. Default RTE Trap Handlers

The default RTE trap handlers are executed when a certain trap is triggered that is not (yet) handled by an application-defined trap handler. The default handler will output a message giving additional debug information via the Primary Universal Asynchronous Receiver and Transmitter (UART0) to inform the user and it will also try to resume normal program execution (exemplary RTE outputs are shown below). The specific message right at the beginning of the debug trap handler message corresponds to the trap code obtained from the `mcause` CSR (see NEORV32 Trap Listing).

In most cases the RTE can successfully continue operation - for example if it catches an **interrupt** request that is not handled by the actual application program. However, if the RTE catches an unhandled **trap** like a bus access fault exception, continuing execution will most likely fail making the CPU crash.

Listing 22. RTE Default Trap Handler UART0 Output Examples

```
<NEORV32-RTE> [cpu0] [M] Illegal instruction @ PC=0x000002d6, MTINST=0x000000FF,
MTVAL=0x00000000 </NEORV32-RTE> ①
<NEORV32-RTE> [cpu0] [U] Illegal instruction @ PC=0x00000302, MTINST=0x00000000,
MTVAL=0x00000000 </NEORV32-RTE> ②
<NEORV32-RTE> [cpu0] [U] Load address misaligned @ PC=0x00000440, MTINST=0x01052603,
MTVAL=0x80000101 </NEORV32-RTE> ③
<NEORV32-RTE> [cpu1] [M] Fast IRQ 0x3 @ PC=0x00000820, MTINST=0x00000000,
MTVAL=0x00000000 </NEORV32-RTE> ④
<NEORV32-RTE> [cpu1] [M] Instruction access fault @ PC=0x90000000, MTINST=0x42078b63,
MTVAL=0x00000000 !!FATAL EXCEPTION!! Halting CPU. </NEORV32-RTE>\n ⑤
```

① Illegal 32-bit instruction `MTINST=0x000000FF` at address `PC=0x000002d6` while the CPU 0 was in machine-mode (`[M]`).

② Illegal 16-bit instruction `MTINST=0x00000000` at address `PC=0x00000302` while the CPU 0 was in user-mode (`[U]`).

③ Misaligned load access at address PC=0x00000440 caused by instruction MTINST=0x01052603 (trying to load a full 32-bit word from address MTVAL=0x80000101) while the CPU 0 was in user-mode ([U]).

④ Fast interrupt request from channel 3 before executing instruction at address PC=0x00000820 while the CPU 1 was in machine-mode ([M]).

⑤ Instruction bus access fault at address PC=0x90000000 while executing instruction MTINST=0x42078b63 while the CPU 1 was in machine-mode ([M]).

4.10.4. Application Context Handling

Upon trap entry the RTE backups the entire application context (i.e. all x general purpose registers) to the stack. The context is restored automatically after trap completion. The base address of the according stack frame is copied to the **mscratch** CSR. By having this information available, the RTE provides dedicated functions for accessing and altering the application context:

Listing 23. RTE Context Access Functions

```
// Prototypes
uint32_t neorv32_rte_context_get(int x); // read register
void     neorv32_rte_context_put(int x, uint32_t data); // write data to register

// Examples
uint32_t tmp = neorv32_rte_context_get(9); // read register 'x9'
neorv32_rte_context_put(28, tmp); // write 'tmp' to register 'x28'
```

The x argument is used to specify one of the RISC-V general purpose register x0 to x31. Note that registers x16 to x31 are not available if the RISC-V E ISA Extension is enabled. For he SMP Dual-Core Configuration the provided context functions will access the stack frame of the interrupted application code that was running on the specific CPU core that caused the trap entry.

The context access functions can be used by application-specific trap handlers to *emulate* unsupported CPU / SoC features like unimplemented IO modules, unsupported instructions and even unaligned memory accesses.

Demo Program: Emulate Unaligned Memory Access

A demo program, which showcases how to emulate unaligned memory accesses using the NEORV32 runtime environment can be found in sw/example/demo_emulate_unaligned.

Chapter 5. On-Chip Debugger (OCD)

The NEORV32 Processor features an *on-chip debugger* (OCD) compatible to the **Minimal RISC-V Debug Specification** implementing the **execution-based debugging** scheme. A copy of the specification is available in `docs/references`. The on-chip debugger is implemented if the `OCD_EN` processor top generic is set to `true`.

Key Features

- standard 4-wire JTAG access port
- debugging of up to 4 CPU cores ("harts")
- full control of the CPU: halting, single-stepping and resuming
- indirect access to all core registers and the entire processor address space (via program buffer)
- execution of arbitrary programs via the program buffer
- compatible with upstream OpenOCD and GDB
- optional trigger module for hardware breakpoints
- optional authentication for increased security

Hands-On Tutorial

A simple example on how to use NEORV32 on-chip debugger in combination with OpenOCD and the GNU debugger is shown in section Debugging using the On-Chip Debugger of the User Guide.

Overview

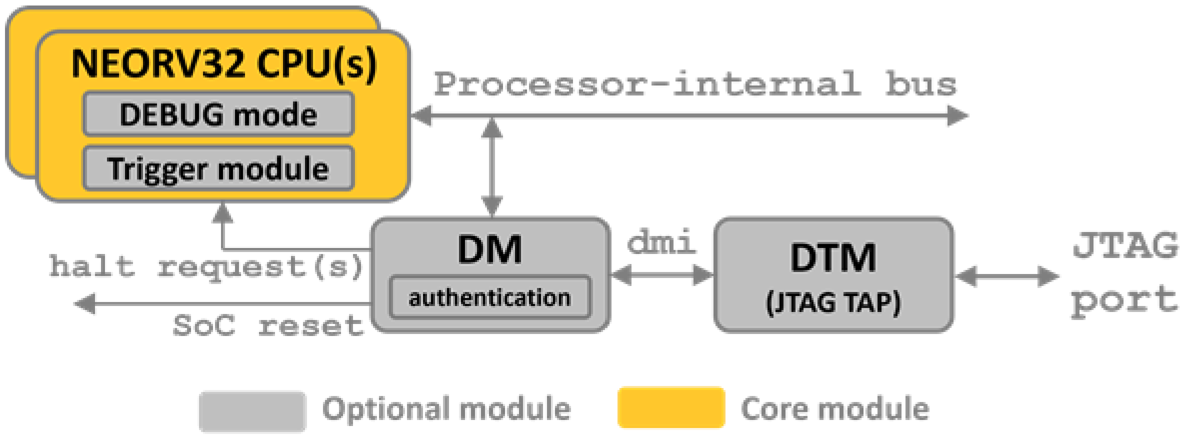


Figure 13. NEORV32 on-chip debugger complex

The NEORV32 on-chip debugger is based on five hardware modules:

1. Debug Transport Module (DTM): JTAG access tap to allow an external adapter to interface with the *debug module (DM)*.
2. Debug Module (DM): The RISC-V debug module is the main bridge between the external

debugger and the processor being debugged. It provides a *data buffer* for data transfer from/to the DM, a *code ROM* containing the "park loop" code, a *program buffer* to allow the debugger to execute small programs defined by the DM and a *status register* that is used to communicate *exception*, *halt*, *resume* and *execute* requests/acknowledges between the debugger and the CPU.

3. Debug Authentication: Authenticator module to secure on-chip debugger access. By default this module implements a very simple authentication mechanism as example. Users can modify/replace this default logic to implement arbitrary authentication mechanism.
4. CPU Debug Mode ISA extension: This ISA extension provides the "debug execution mode" as another CPU operation mode that is used to execute the park loop code from the DM. This mode also provides additional CSRs and instructions.
5. CPU Trigger Module: This module provides a single *hardware breakpoint*.

Theory of Operation

When debugging the system using the OCD, the external debugger (e.g. GDB) issues a halt request to the CPU to make it enter so-called *debug mode*. In this mode the application-defined architectural state of the system/CPU is "frozen" so the debugger can monitor it without interfering with the actual application. However, the OCD can also modify the entire architectural state at any time. While in debug mode, the debugger has full control over the entire CPU core.

After halting, the CPU executes the "park loop" code from the code ROM of the debug module (DM). This park loop implements an endless loop that is used to poll a memory-mapped Status Register of the DM. The flags in this register are used to communicate requests from the DM and to acknowledge their processing them by the CPU: trigger execution of the program buffer or resume the halted application. Furthermore, the CPU uses this register to signal that the CPU has halted after a halt request or to signal that an exception has been raised while being in debug mode.

5.1. Debug Transport Module (DTM)

The debug transport module "DTM" (VHDL module: `rtl/core/neorv32_debug_dtm.vhd`) provides a bridge between a standard 4-wire JTAG test access port ("tap") and the internal debug module interface.

Table 95. JTAG Top Level Signals of the DTM

Name	Width	Direction	Description
`jtag_tck_i`	1	in	serial clock
`jtag_tdi_i`	1	in	serial data input
`jtag_tdo_o`	1	out	serial data output
`jtag_tms_i`	1	in	mode select

Maximum JTAG Clock

All JTAG signals are synchronized to the processor's clock domain. Hence, no additional clock domain is required for the DTM. However, this constraints the maximal JTAG clock frequency (`jtag_tck_i`) to be less than or equal to **1/5** of the processor clock frequency (`clk_i`).

JTAG TAP Reset

The NEORV32 JTAG TAP does not provide a dedicated reset signal ("TRST"). However, JTAG-level resets can be triggered using TMS signaling.

Maintaining the JTAG Chain

If the on-chip debugger is disabled the JTAG serial input `jtag_tdi_i` is directly connected to the JTAG serial output `jtag_tdo_o` to maintain the JTAG chain.

The DTM implement a single 5-bit *instruction register* `IR` and several *data registers* `DR` with different sizes. The individual data registers are accessed by writing the according address to the instruction register. The following table shows all available data registers and their addresses:

Table 96. JTAG TAP registers

Address (via `IR`)	Name	Size (bits)	Description
`00001`	`IDCODE`	32	identification code (see below)
`10000`	`DTMCS`	32	debug transport module control and status register (see below)
`10001`	`DMI`	41	debug module interface (see below)
others	`BYPASS`	1	default JTAG bypass register

Table 97. `IDCODE` - DTM Identification Code Register

Bit(s)	Name	R/W	Description
31:28	`version`	r/-	version ID, hardwired to zero
27:12	`partid`	r/-	part ID, hardwired to zero
11:1	`manid`	r/-	JEDEDC manufacturer ID, assigned via the `JEDEC_ID` generic
0	-	r/-	hardwired to `1`

Table 98. `DTMCS` *- DTM Control and Status Register*

Bit(s)	Name	R/W	Description
31:18	-	r/-	*reserved*, hardwired to zero
17	`dmihardreset`	r/w	setting this bit will reset the debug module interface; this bit auto-clears
16	`dmireset`	r/w	setting this bit will clear the sticky error state; this bit auto-clears
15	-	r/-	*reserved*, hardwired to zero
14:12	`idle`	r/-	recommended idle states (= 0, no idle states required)
11:10	`dmistat`	r/-	DMI status: `00` = no error, `01` = reserved, `10` = operation failed, `11` = failed operation during pending DMI operation
9:4	`abits`	r/-	number of address bits in `DMI` register (= 6)
3:0	`version`	r/-	`0001` = DTM is compatible to RISC-V debug spec. versions v0.13 and v1.0

Table 99. `DMI` *- DTM Debug Module Interface Register*

Bit(s)	Name	R/W	Description
40:34	`address`	r/w	7-bit address, see DM Registers
33:2	`data`	r/w	32-bit to write/read to/from the addresses DM register
1:0	`command`	r/w	2-bit operation (`00` = NOP; `10` = write; `01` = read)

5.2. Debug Module (DM)

The debug module "DM" (VHDL module: `rtl/core/neorv32_debug_dm.vhd`) acts as a translation interface between abstract operations issued by the debugger application (like GDB) and the platform-specific debugger hardware. It supports the following features:

- Gives the debugger necessary information about the implementation.
- Allows the hart to be halted/resumed/reset and provides the current status.
- Provides abstract read and write access to the halted hart's general purpose registers.
- Provides access to a reset signal that allows debugging from the very first instruction after reset.
- Provides a *program buffer* to force the hart to execute arbitrary instructions.
- Allows memory accesses (to the entire address space) from a hart's point of view.
- Optionally implements an authentication mechanism to secure on-chip debugger access.

The NEORV32 DM follows the "Minimal RISC-V External Debug Specification" to provide full debugging capabilities while keeping resource/area requirements at a minimum. It implements the **execution based debugging scheme** for up to four individual CPU cores ("harts") and provides the following architectural core features:

- program buffer with 2 entries and an implicit `ebreak` instruction at the end
- indirect bus access via the CPU using the program buffer
- abstract commands: "access register" plus auto-execution
- halt-on-reset capability
- optional authentication

DM Spec. Version

The NEORV32 DM complies to the RISC-V DM spec version 1.0.

From the DTM's point of view, the DM implements a set of DM Registers that are used to control and monitor the debugging session. From the CPU's point of view, the DM implements several memory-mapped registers that are used for communicating data, instructions, debugging control and status (DM CPU Access).

External Reset Output

The entire processor can be reset at any time by the debugger via the `ndmreset` bit of the **`dmcontrol`** register. This signal is also available as processor top signal (Processor Top Entity - Signals: `rstn_ocd_o`) and can be used to reset processor-external modules via the on-chip debugger. This signal is low-active and synchronous to the processor clock. It is available if the on-chip debugger is actually implemented; otherwise it is hardwired to `1`. Note that the signal also becomes active (low) when the processor's main reset signal is active (even if the on-chip debugger is deactivated or disabled for synthesis).

5.2.1. DM Registers

The DM is controlled via a set of registers that are accessed via the DTM. The following registers are implemented:

Unimplemented Registers

Write accesses to registers that are not implemented are simply ignored and read accesses to these registers will always return zero. In both cases no error condition is signaled to the DTM.

Table 100. Available DM registers

Address	Name	Description
0x04	`data0`	Abstract data register 0
0x10	`dmcontrol`	Debug module control
0x11	`dmstatus`	Debug module status
0x12	`hartinfo`	Hart information
0x16	`abstracts`	Abstract control and status
0x17	`command`	Abstract command
0x18	`abstractauto`	Abstract command auto-execution
0x1d	`nextdm`	Base address of next DM; reads as zero to indicate there is only one DM
0x20	`progbuf0`	Program buffer 0
0x21	`progbuf1`	Program buffer 1
0x30	`authdata`	Data to/from the authentication module
0x38	`sbcs`	System bus access control and status; reads as zero to indicate there is **no** system bus access
0x40	`haltsum0`	Hart halt summary

`data0`

0x04 **Abstract data 0** `data0`

Reset value: `0x00000000`

Basic read/write data exchange register to be used with abstract commands (for example to read/write data from/to CPU GPRs).

`dmcontrol`

0x10 **Debug module control register** `dmcontrol`

Reset value: `0x00000000`

Control of the overall debug module and the hart. The following table shows all implemented bits. All remaining bits/bit-fields are configured as "zero" and are read-only. Writing '1' to these bits/fields will be ignored.

Table 101. `dmcontrol` Register Bits

Bit	Name [RISC-V]	R/W	Description
31	`haltreq`	-/w	set/clear hart halt request
30	`resumereq`	-/w	request hart to resume
28	`ackhavereset`	-/w	write `1` to clear `*havereset` flags
27	-	r/-	reserved, hardwired to zero
26	`hasel`	r/-	`0`: only a single hart can be selected at once
25:16	`hartsello`	r/w	hart select; only the lowest 3 bits are implemented
15:6	`hartselhi`	r/-	hardwired to zero
5:4	-	r/-	reserved, hardwired to zero
3	`setresethaltreq`	r/-	`0`: halt-on-reset not implemented
2	`clrresethaltreq`	r/-	`0`: halt-on-reset not implemented
1	`ndmreset`	r/w	put whole system (except OCD) into reset state when `1`
0	`dmactive`	r/w	DM enable; writing `0`-`1` will reset the DM

`dmstatus`

0x11 **Debug module status register** `dmstatus`

Reset value: `0x00400083`

Current status of the overall debug module and the hart. The entire register is read-only.

Table 102. `dmstatus` Register Bits

Bit	Name [RISC-V]	Description
31:23	*reserved*	reserved; zero
22	`impebreak`	`1`: indicates an implicit `ebreak` instruction after the last program buffer entry
21:20	*reserved*	reserved; zero
19	`allhavereset`	`1` when the selected hart is in reset state
18	`anyhavereset`	
17	`allresumeack`	`1` when the selected hart has acknowledged a resume request
16	`anyresumeack`	

Bit	Name [RISC-V]	Description
15	`allnonexistent`	`1` when the selected hart is not available
14	`anynonexistent`	
13	`allunavail`	`1` when the DM is disabled to indicate the selected hart is unavailable
12	`anyunavail`	
11	`allrunning`	`1` when the selected hart is running
10	`anyrunning`	
9	`allhalted`	`1` when the selected hart is halted
8	`anyhalted`	
7	`authenticated`	set if authentication passed; see Debug Authentication
6	`authbusy`	set if authentication is busy, see Debug Authentication
5	`hasresethaltreq`	`0`: halt-on-reset is not supported (directly)
4	`confstrptrvalid`	`0`: no configuration string available
3:0	`version`	`0011`: DM compatible to debug spec. version v1.0

`hartinfo`

0x12 **Hart information** `hartinfo`

Reset value: *see below*

This register gives information about the hart. The entire register is read-only.

Table 103. `hartinfo` Register Bits

Bit	Name [RISC-V]	Description
31:24	*reserved*	reserved; zero
23:20	`nscratch`	`0001`: number of `dscratch*` CPU registers = 1
19:17	*reserved*	reserved; zero
16	`dataccess`	`0`: the `data` registers are shadowed in the hart's address space
15:12	`datasize`	`0001`: number of 32-bit words in the address space dedicated to shadowing the `data` registers (1 register)
11:0	`dataaddr`	= `dm_data_base_c(11:0)`, signed base address of `data` words (see address map in DM CPU Access)

abstracts

0x16 **Abstract control and status** abstracts

Reset value: `0x02000801`

Command execution info and status.

Table 104. `abstracts` Register Bits

Bit	Name [RISC-V]	R/W	Description
31:29	*reserved*	r/-	reserved; zero
28:24	`progbufsize`	r/-	`0010`: size of the program buffer (`progbuf`) = 2 entries
23:11	*reserved*	r/-	reserved; zero
12	`busy`	r/-	set when a command is being executed
11	`relaxedpriv`	r/-	`1`: PMP rules are ignored when in debug mode
10:8	`cmderr`	r/w	error during command execution (see below); has to be cleared by writing `111`
7:4	*reserved*	r/-	reserved; zero
3:0	`datacount`	r/-	`0001`: number of implemented `data` registers for abstract commands = 1

Error codes in `cmderr` (highest priority first):

* `000` - no error
* `100` - command cannot be executed since hart is not in expected state
* `011` - exception during command execution
* `010` - unsupported command
* `001` - invalid DM register read/write while command is/was executing

command

0x17 **Abstract command** command

Reset value: `0x00000000`

Writing this register will trigger the execution of an abstract command. New command can only be executed if `cmderr` is zero. The entire register in write-only (reads will return zero).

The NEORV32 DM only supports **Access Register** abstract commands. These commands can only access the hart's GPRs x0 - x15/31 (abstract command register index `0x1000` - `0x101f`).

Table 105. `command` Register Bits

Copyright by Stephan Nolting. All rights reserved.

Bit	Name [RISC-V]	R/W	Description / required value
31:24	`cmdtype`	-/w	`00000000`: indicates "access register" command
23	*reserved*	-/w	reserved, has to be `0` when writing
22:20	`aarsize`	-/w	`010`: indicates 32-bit accesses
21	`aarpostincrement`	-/w	`0`: post-increment is not supported
18	`postexec`	-/w	set if the program buffer is executed *after* the command
17	`transfer`	-/w	set if the operation in `write` is conducted
16	`write`	-/w	`1`: copy `data0` to `[regno]`, `0`: copy `[regno]` to `data0`
15:0	`regno`	-/w	GPR-access only; has to be `0x1000` - `0x101f`

`abstractauto`

0x18 **Abstract command auto-execution** `abstractauto`

Reset value: `0x00000000`

Register to configure if a read/write access to a DM register re-triggers execution of the last abstract command.

Table 106. `abstractauto` Register Bits

Bit	Name [RISC-V]	R/W	Description
17	`autoexecprogbuf[1]`	r/w	when set reading/writing from/to `progbuf1` will execute `command` again
16	`autoexecprogbuf[0]`	r/w	when set reading/writing from/to `progbuf0` will execute `command` again
0	`autoexecdata[0]`	r/w	when set reading/writing from/to `data0` will execute `command` again

`progbuf`

0x20 **Program buffer 0** `progbuf0`

0x21 **Program buffer 1** `progbuf1`

Reset value: `0x00000013` ("NOP")

Program buffer (two entries) for the DM.

`authdata`

0x30 **Authentication data** `authdata`

Reset value: *user-defined*

This register serves as a 32-bit serial port to/from the authentication module. See Debug Authentication.

haltsum0

0x30	**Halt summary 0**	haltsum0

Reset value: 0x00000000

Each bit corresponds to a hart being halted. Only the lowest four bits are implemented.

5.2.2. DM CPU Access

From the CPU's perspective the DM acts like another memory-mapped peripheral. It occupies 512 bytes of the CPU's address space starting at address base_io_dm_c (0xffff0000). This address space is divided into four sections 128 64 bytes each to provide access to the *park loop code ROM*, the *program buffer*, the *data buffer* and the *status register*. The program buffer, the data buffer and the status register do not fully occupy the 128-byte-wide sections and are mirrored several times across the entire section.

Table 107. DM CPU Access - Address Map

Base address	Physical size	Description
0xfffffe00	128 bytes	ROM for the "park loop" code (Code ROM)
0xfffffe80	16 bytes	Program buffer (**progbuf**)
0xffffff00	4 bytes	Data buffer (**data0**)
0xffffff80	16 bytes	Control and Status Register

DM Register Access

All memory-mapped registers of the DM can only be accessed by the CPU when in debug mode. Hence, the DM registers are not accessible for normal CPU operations. Any CPU access outside of debug mode will raise a bus access fault exception.

Code ROM

The code ROM contain the minimal OCD firmware that implements the debuggers part loop.

Park Loop Code Sources ("OCD Firmware")

The assembly sources of the park loop code are available in sw/ocd-firmware/park_loop.S.

The park loop code provides two entry points where code execution can start. These are used to enter the park loop either when an explicit debug-entry/halt request has been issued (for example a halt request) or when an exception has occurred while executing code in debug mode (from the profram buffer).

Table 108. Park Loop Entry Points

Address	Description
`dm_exc_entry_c` (`base_io_dm_c` + 0)	Exception entry address
`dm_park_entry_c` (`base_io_dm_c` + 16)	Normal entry address (halt request)

When the CPU enters (via an explicit halt request from the debugger) or re-enters debug mode (for example via an `ebreak` in the DM's program buffer), it jumps to the **normal entry point** that is configured via the `CPU_DEBUG_PARK_ADDR` CPU generic. By default, this address is set to `dm_park_entry_c`, which is defined in the main package file. If an exception is encountered during debug mode, the CPU jumps to the address of the **exception entry point** configured via the `CPU_DEBUG_EXC_ADDR` CPU generic. By default, this address is set to `dm_exc_entry_c`, which is also defined in the main package file.

Status Register

The status register provides a direct communication channel between the CPU's debug-mode executing the park loop and the debugger-controlled DM. This register is used to communicate requests, which are issued by the DM, and the according acknowledges, which are generated by the CPU. The status register is sub-divided into four consecutive memory-mapped registers.

Starting at `0xffffff80` the status register provides a set of memory-mapped interface register whose functionality depends on whether the CPU accesses the register in read or write mode. **Read** accesses return the **requests** for each individual hart generated by the DM. **Write** accesses are used to **acknowledge** these requests by the individual harts back to the DM.

For read accesses, the hart ID is used as byte offset to read the hart-specific request flags. The flags for hart 0 are located at `0xffffff80 + 0`, the flags for hart 1 are located at `0xffffff80 + 1` and so on. Hence, each hart can use load-unsigned-byte instructions to isolate the hart specific flags.

Table 109. DM Status Register - Read Access (byte-wise access)

Address	Hart	R/W	Bits	Description
`0xffffff80`	0	r/-	0	Resume request
			1	Execute request
`0xffffff81`	1	r/-	0	Resume request
			1	Execute request
`0xffffff82`	2	r/-	0	Resume request
			1	Execute request
`0xffffff83`	3	r/-	0	Resume request
			1	Execute request

For write accesses, four consecutive memory-mapped registers are implemented. Each individual register is used to acknowledge a specific condition: halt, resume, execute and exception. Each hart can acknowledge the according condition by writing its hart ID to the according register.

Table 110. DM Status Register - Write Access (word-wise access)

Address	R/W	Bits	Description
`0xffffff80`	r/w	1:0	write hart ID to send hart's HALT acknowledge
`0xffffff84`	r/w	1:0	write hart ID to send hart's RESUME acknowledge
`0xffffff88`	r/w	1:0	write hart ID to send hart's EXECUTE acknowledge
`0xffffff8c`	r/w	1:0	write any value to send hart's EXCEPTION acknowledge

5.3. Debug Authentication

Optionally, the on-chip debugger's DM can be equipped with an *authenticator module* to secure debugger access. This authentication is enabled by the `OCD_AUTHENTICATION` top generic. When disabled, the debugger is always authorized and has unlimited access. When enabled, the debugger is required to authenticate in order to gain access.

The authenticator module is implemented as individual RTL module (`rtl/core/neorv32_debug_auth.vhd`). By default, it implements a very simple authentication mechanism. Note that this default mechanism is not secure in any way - it is intended as example logic to illustrate the interface and authentication process. Users can modify the default logic or replace the entire module to implement a more sophisticated custom authentication mechanism.

The authentication interface is compliant to the RISC-V debug spec and is based on a single CSR and two additional status bits:

- **`authdata`** CSR: this 32-bit register is used to read/write data from/to the authentication module. It is hardwired to all-zero if authentication is not implemented.
- **`dmstatus`** CSR:
  - The `authenticated` bit (read-only) is set if authentication was successful. The debugger can access the processor only if this bit is set. It is automatically hardwired to `1` (always authenticated) if the authentication module is not implemented.
  - The `authbusy` bit (read-only) indicates if the authentication module is busy. When set, no data should be written/read to/from **`authdata`**. This bit is automatically hardwired to `0` (never busy) if the authentication module is not implemented.

openOCD provides dedicated commands to exchange data with the authenticator module:

Listing 24. openOCD RISC-V Authentication Commands

```
riscv authdata_read        // read 32-bit from authdata CSR
riscv authdata_write value // write 32-bit value to authdata CSR
```

Based on these two primitives arbitrary complex authentication mechanism can be implemented.

5.3.1. Default Authentication Mechanism

The default authentication mechanism is not secure at all. Replace it by a custom design.

The default authenticator hardware implements a very simple authentication mechanism: a single read/write bit is implemented that directly corresponds to the `authenticated` bit in **`dmstatus`**. This bit can be read/written as bit zero (LSB) of the **`authdata`** CSR. Writing 1 to this register will result in a successful authentication. The default openOCD configuration script for the NEORV32 implements this basic authentication mechanism:

Listing 25. Default authentication process (`sw/openocd/openocd_neorv32.cfg`)

```
set challenge [riscv authdata_read]          # read authdata; not required, just an
example
riscv authdata_write [expr {$challenge | 1}] # set LSB to authenticate
```

5.4. CPU Debug Mode

The NEORV32 CPU Debug Mode is compatible to the **Minimal RISC-V Debug Specification 1.0** `Sdext` (external debug) ISA extension. When enabled via the CPU's `Sdext` ISA Extension generic and/or the processor's `OCD_EN` it adds a new CPU operation mode ("debug mode"), three additional CPU Debug Mode CSRs and one additional instruction (`dret`) to the core.

Debug-mode is entered on any of the following events:

1. The CPU executes an `ebreak` instruction (when in machine-mode and `ebreakm` in **`dcsr`** is set OR when in user-mode and `ebreaku` in **`dcsr`** is set).
2. A debug halt request is issued by the DM (via CPU `db_halt_req_i` signal, high-active).
3. The CPU completes executing of a single instruction while being in single-step debugging mode (`step` in **`dcsr`** is set).
4. A hardware trigger from the Trigger Module fires (`exe` and `action` in **`tdata1`** / `mcontrol` are set).

From a hardware point of view these debug-mode-entry conditions are special traps (synchronous exceptions or asynchronous interrupts) that are handled transparently by the control logic.

Whenever the CPU enters debug-mode it performs the following operations:

- wake-up CPU if it was send to sleep mode by the `wfi` instruction
- switch to debug-mode privilege level
- move the current program counter to **`dpc`**
- copy the hart's current privilege level to the `prv` flags in **`dcsr`**
- set `cause` in **`dcsr`** according to the cause why debug mode is entered
- **no update** of `mtval`, `mcause`, `mtval` and `mstatus` CSRs
- load the address configured via the CPU's (`CPU_DEBUG_PARK_ADDR`) generic to the program counter jumping to the "debugger park loop" code stored in the debug module (DM)

When the CPU is in debug-mode:

- while in debug mode, the CPU executes the parking loop and - if requested by the DM - the program buffer
- effective CPU privilege level is `machine` mode; any active physical memory protection (PMP) configuration is bypassed
- the `wfi` instruction acts as a `nop` (also during single-stepping)
- if an exception occurs while being in debug mode:
  - if the exception was caused by any debug-mode entry action the CPU jumps to the normal entry point (defined by the `CPU_DEBUG_PARK_ADDR` generic) of the park loop again (for example when executing `ebreak` while in debug-mode)

  - for all other exception sources the CPU jumps to the exception entry point (defined by the `CPU_DEBUG_EXC_ADDR` generic) to signal an exception to the DM; the CPU restarts the park loop again afterwards
- interrupts are disabled; however, they will remain pending and will get executed after the CPU has left debug mode and is not being single-stepped
- if the DM makes a resume request, the park loop exits and the CPU leaves debug mode (executing `dret`)
- the standard counters (Machine) Counter and Timer CSRs `[m]cycle[h]` and `[m]instret[h]` are stopped
- all Hardware Performance Monitors (HPM) CSRs are stopped

Debug mode is left either by executing the `dret` instruction or by performing a hardware reset of the CPU. Executing `dret` outside of debug mode will raise an illegal instruction exception.

Whenever the CPU leaves debug mode it performs the following operations:

- set the hart's current privilege level according to the `prv` flags of **`dcsr`**
- restore the original program counter from **`dpc`** resuming normal operation

5.4.1. CPU Debug Mode CSRs

Two additional CSRs are required by the "Minimal RISC-V Debug Specification": the debug mode control and status register `dcsr` and the debug program counter `dpc`. An additional general purpose scratch register for debug-mode-only (`dscratch0`) allows faster execution by having a fast-accessible backup register. These CSRs are only accessible if the CPU is in debug mode. If these CSRs are accessed outside of debug mode an illegal instruction exception is raised.

`dcsr`

Name	Debug control and status register
Address	`0x7b0`
Reset value	`0x40000410`
ISA	`Zicsr` & `Sdext`
Description	This register is used to configure the debug mode environment and provides additional status information.

Table 111. Debug control and status register `dcsr` bits

Bit	Name [RISC-V]	R/W	Description
31:28	`xdebugver`	r/-	`0100`: CPU debug mode is compatible to spec. version 1.0
27:16	-	r/-	`000000000000`: *reserved*

Bit	Name [RISC-V]	R/W	Description
15	`ebereakm`	r/w	`ebreak` instructions in `machine` mode will *enter* debug mode when set
14	`ebereakh`	r/-	`0`: hypervisor mode not supported
13	`ebereaks`	r/-	`0`: supervisor mode not supported
12	`ebereaku`	r/w	`ebreak` instructions in `user` mode will *enter* debug mode when set
11	`stepie`	r/-	`0`: IRQs are disabled during single-stepping
10	`stopcount`	r/-	`1`: standard counters and HPMs are stopped when in debug mode
9	`stoptime`	r/-	`0`: timers increment as usual
8:6	`cause`	r/-	cause identifier: why debug mode was entered (see below)
5	-	r/-	`0`: *reserved*
4	`mprven`	r/-	`1`: `mprv` in **`mstatus`** is also evaluated when in debug mode
3	`nmip`	r/-	`0`: non-maskable interrupt is pending
2	`step`	r/w	enable single-stepping when set
1:0	`prv`	r/w	CPU privilege level before/after debug mode

Cause codes in `dcsr.cause` (highest priority first):

* `010` - triggered by hardware Trigger Module
* `001` - executed `EBREAK` instruction
* `011` - external halt request (from DM)
* `100` - return from single-stepping

dpc

Name	Debug program counter
Address	`0x7b1`
Reset value	`0x00000000`
ISA	`Zicsr` & `Sdext`
Descripti on	The register is used to store the current program counter when debug mode is entered. The `dret` instruction will return to the address stored in `dpc` by automatically moving `dpc` to the program counter.

`dpc[0]` is hardwired to zero. If `IALIGN` = 32 (i.e. `C` ISA Extension is disabled) then `dpc[1]` is also hardwired to zero.

`dscratch0`

Name	Debug scratch register 0
Address	`0x7b2`
Reset value	`0x00000000`
ISA	`Zicsr` & `Sdext`
Description	The register provides a general purpose debug mode-only scratch register.

5.5. Trigger Module

"Normal" software breakpoints (using GDB's `b`/`break` command) are implemented by temporarily replacing the according instruction word by an `[c.]ebreak` instruction. However, this is not possible when debugging code that is executed from read-only memory (for example when debugging programs that are executed via the Execute In Place Module (XIP)). To circumvent this limitation a hardware trigger logic allows to (re-)enter debug-mode when instruction execution reaches a programmable address. These "hardware-assisted breakpoints" are used by GDB's `hb`/`hbreak` commands.

The RISC-V `Sdtrig` ISA extension adds a programmable *trigger module* to the CPU core that is enabled via the `Sdtrig` ISA Extension generic. The trigger module implements a subset of the features described in the "RISC-V Debug Specification / Trigger Module" and complies to version v1.0 of the `Sdtrig` spec.

The NEORV32 trigger module features only a *single* trigger implementing a "type 6 - instruction address match" trigger. This limitation is granted by the RISC-V debug spec and is sufficient to **debug code executed from read-only memory (ROM)**. The trigger module can also be used independently of the CPU debug-mode / `Sdext` ISA extension. Machine-mode software can use the trigger module to raise a breakpoint exception when instruction execution reaches a programmed address.

Trigger Timing

When enabled the address match trigger will fire **BEFORE** the instruction at the programmed address gets executed.

MEPC & DPC CSRs

The breakpoint exception when raised by the trigger module behaves different then the "normal" trapping (see NEORV32 Trap Listing): **`mepc`** / **`dpc`** is set to the address of the next instruction that needs to be executed to preserve the program flow. A "normal" breakpoint exception would set **`mepc`** / **`dpc`** to the address of the actual `ebreak` instruction itself.

5.5.1. Trigger Module CSRs

The `Sdtrig` ISA extension adds 4 additional CSRs that are accessible from debug-mode and also from machine-mode. Machine-mode write accesses can be ignored by setting ´dmode´ in **`tdata1`**. This is automatically done by the debugger if it uses the trigger module for implementing a "hardware breakpoint"

`tselect`

Name	Trigger select register
Address	`0x7a0`

Reset value	`0x00000000`
ISA	`Zicsr` & `Sdtrig`
Description	This CSR is hardwired to zero indicating there is only one trigger available. Any write access is ignored.

`tdata1`

Name	Trigger data register 1, visible as trigger "type 6 match control" (`mcontrol6`)
Address	`0x7a1`
Reset value	`0x60000048`
ISA	`Zicsr` & `Sdtrig`
Description	This CSR is used to configure the address match trigger using "type 6" format.

Table 112. Match Control CSR (`tdata1`) Bits

Bit	Name [RISC-V]	R/W	Description
31:28	`type`	r/-	`0100`: address match trigger type 6
27	`dmode`	r/w	set to ignore write accesses to **`tdata1`** and **`tdata2`** from machine-mode; writable from debug-mode only
26	`uncertain`	r/-	`0`: trigger satisfies the configured conditions
25	`hit1`	r/-	`0`: hardwired to zero, only `hit0` is used
24	`vs`	r/-	`0`: VS-mode not supported
23	`vu`	r/-	`0`: VU-mode not supported
22	`hit0`	r/c	set when trigger has fired (**BEFORE** executing the triggering address); must be explicitly cleared by writing zero; writing 1 has no effect
21	`select`	r/-	`0`: only address matching is supported
20:19	reserved	r/-	`00`: hardwired to zero
18:16	`size`	r/-	`000`: match accesses of any size
15:12	`action`	r/w	`0000` = breakpoint exception on trigger match, `0001` = enter debug-mode on trigger match
11	`chain`	r/-	`0`: chaining is not supported as there is only one trigger
10:6	`match`	r/-	`0000`: equal-match only
6	`m`	r/-	`1`: trigger enabled when in machine-mode
5	`uncertainen`	r/-	`0`: feature not supported, hardwired to zero

Bit	Name [RISC-V]	R/W	Description
4	`s`	r/-	`0`: supervisor-mode not supported
3	`u`	r/-	`0`/`1`: trigger enabled when in user-mode, set if `U` ISA extension is enabled
2	`execute`	r/w	set to enable trigger matching on instruction address
1	`store`	r/-	`0`: store address/data matching not supported
0	`load`	r/-	`0`: load address/data matching not supported

`tdata2`

Name	Trigger data register 2
Address	`0x7a2`
Reset value	`0x00000000`
ISA	`Zicsr` & `Sdtrig`
Description	Since only the "address match trigger" type is supported, this r/w CSR is used to configure the address of the triggering instruction. Note that the trigger module will fire **before** the instruction at the programmed address gets executed.

`tinfo`

Name	Trigger information register
Address	`0x7a4`
Reset value	`0x01000006`
ISA	`Zicsr` & `Sdtrig`
Description	The CSR shows global trigger information (see below). Any write access is ignored.

Table 113. Trigger Info CSR (`tinfo`) Bits

Bit	Name [RISC-V]	R/W	Description
31:24	`version`	r/-	`0x01`: compatible to spec. version v1.0
23:15	reserved	r/-	`0x00`: hardwired to zero
15:0	`info`	r/-	`0x0006`: only "type 6 trigger" is supported

Chapter 6. Legal

About

The NEORV32 RISC-V Processor
https://github.com/stnolting/neorv32
Stephan Nolting, M.Sc.
🇪🇺 European Union
stnolting@gmail.com

License

BSD 3-Clause License

Copyright (c) NEORV32 contributors. Copyright (c) 2020 - 2025, Stephan Nolting. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

SPDX Identifier

`SPDX-License-Identifier: BSD-3-Clause`

Proprietary Notice

- "GitHub" is a subsidiary of Microsoft Corporation.
- "Vivado" and "Artix" are trademarks of AMD Inc.
- "AXI", "AXI", "AXI4-Lite", "AXI4-Stream", "AHB", "AHB3" and "AHB3-Lite" are trademarks of Arm Holdings plc.
- "ModelSim" is a trademark of Mentor Graphics – A Siemens Business.
- "Quartus [Prime]" and "Cyclone" are trademarks of Intel Corporation.
- "iCE40", "UltraPlus" and "Radiant" are trademarks of Lattice Semiconductor Corporation.
- "GateMate" is a trademark of Cologne Chip AG.
- "Windows" is a trademark of Microsoft Corporation.
- "Tera Term" copyright by T. Teranishi.
- "NeoPixel" is a trademark of Adafruit Industries.
- "Segger Embedded Studio" and "J-Link" are trademarks of Segger Microcontroller Systems GmbH.
- Images/figures made with *Microsoft Power Point*.
- Timing diagrams made with *WaveDrom Editor*.
- Documentation made with `asciidoctor`.

All further/unreferenced projects/products/brands belong to their according copyright holders. No copyright infringement intended.

Disclaimer

This project is released under the BSD 3-Clause license. NO COPYRIGHT INFRINGEMENT INTENDED. Other implied or used projects/sources might have different licensing – see their according documentation for more information.

Limitation of Liability for External Links

This document contains links to the websites of third parties ("external links"). As the content of these websites is not under our control, we cannot assume any liability for such external content. In all cases, the provider of information of the linked websites is liable for the content and accuracy of the information provided. At the point in time when the links were placed, no infringements of the law were recognizable to us. As soon as an infringement of the law becomes known to us, we will immediately remove the link in question.

Citing

This is an open-source project that is free of charge. Use this project in any way you like (as long as it complies to the permissive license). Please cite it appropriately. □

Contributors & Community □

Please add as many contributors as possible to the `author` field.
This project would not be where it is without them.

DOI

This project provides a *digital object identifier* provided by zenodo: DOI 10.5281/zenodo.5018888

Acknowledgments

A big shout-out to the community and all the contributors, who helped improving this project! This project would not be where it is without them. □□

RISC-V - instruction sets want to be free!

Continuous integration provided by GitHub Actions and powered by GHDL.